DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TOWARD HIGHLY PORTABLE DATABASE SYSTEMS:
ISSUES AND SOLUTIONS

by

Albert Wong

June 1986

Thesis Advisor: David K. Hsiao

Approved for public release; distribution unlimited

T233155

CURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	S MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School 6b OFFICE SYMBOL (If applicable): Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
TOWARD HIGHLY PORTABLE DATABASE SYSTEMS: ISSUES AND SOLUTIONS

PERSONAL AUTHOR(S)
Albert Wong

a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	1S PAGE COUNT
Masters Thesis	FROM _____ TO _____	June 1986	93

SUPPLEMENTARY NOTATION

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Software Portability, Software Engineering, Database Systems, Data Models, Query Languages, Record Templates, Network Communications, Disk Input/Output

ABSTRACT (Continue on reverse if necessary and identify by block number)
The multi-backend database system MBDS) is a database system of two or more processors and their dedicated isk subsystems. One of the processors serves as a controller. The rest of he processors and their disks serves as backends to provide the primary and arallel database operations. User access to the MBDS is accomplished ither via a host computer which in turn communicates with the controller, r with the MBDS controller directly. The thesis is aimed to examine the ortability of MBDS. By downloading the MBDS software from the configuration of VAX and PDP hardware and VMS and RSX operating systems to the onfiguration of the 32-bit microprocessor-based ISI hardware and UNIX perating system, we hope to determine the necessary amount of hardware-nd-operating-system-dependent modifications and reinstrumentations in order o make the downloading successful. The ultimate goal of the thesis is to ecommend to the future database-system designer the way to minimize the

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
a NAME OF RESPONSIBLE INDIVIDUAL David K. Hsiao	22b TELEPHONE (Include Area Code) (408) 646-2253 22c OFFICE SYMBOL 52Hq

D FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

amount of configuration-dependent software and to strive for a truly and highly portable system to be used on various configurations. This thesis has identified three major portability issues and provided solutions to them. They are the multiple-record template support, the interprocess communications via broadcasting, and the disk I/O for the real-time access.

Approved for public release. distribution is unlimited.

Towards Highly Portable Database Systems: Issues and Solutions

by

Albert Wong
B. S., West Coast University, 1966

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1986

ABSTRACT

The multi-backend database system (MBDS) is a database system of two or more processors and their dedicated disk subsystems. One of the processor serves as a controller. The rest of the processors and their disks serves as backends to provide the primary and parallel database operations. User access to the MBDS is accomplished either via a host computer which in turn communicates with the controller, or with the MBDS controller directly.

The thesis is aimed to examine the portability of MBDS. By downloading the MBDS software from the configuration of VAX and PDP hardware and VMS and RSX operating systems to the configuration of the 32-bit microprocessor-based ISI hardware and UNIX operating system, we hope to determine the necessary amount of hardware-and-operating-system-dependent modifications and reinstrumentations in order to make the downloading successful. The ultimate goal of the thesis is to recommend to the future database-system designer the way to minimize the amount of configuration-dependent software and to strive for a truly and highly portable system to be used on various configurations. This thesis has identified three major portability issues and provided solutions to them. They are the multiple-record template support. the interprocess communication via broadcasting, and the disk I/O for the real-time access.

TABLE OF CONTENTS

I. AN INTRODUCTION 7

A. THE INTENT OF THE THESIS 8

B. SOFTWARE PORTABILITY ISSUES 9

C. THE ORGANIZATION OF THE THESIS 10

II. MULTIPLE-RECORD TEMPLATES 12

A. THE DAT.. MODEL AND DATA LANGUAGE 13

1. The Attribute-Based Data Model 13

2. The Attribute-Based Language (ABDL) 15

B. SPECIFICATIONS OF NEW TEMPLATES 17

1. Template Descriptions 18

2. Descriptor Specifications 21

3. Database Records 23

C. LOADING THE DATABASE 25

1. Record Handling 25

2. The Record Clustering and Placement 29

III. THE MESSAGE PASSING FACILITIES 31

A. THE ISI-WORKSTATION CONFIGURATION 32

1. The Hardware Organization 31

2. The Software Environment 35

B. THE MBDS PROCESS STRUCTURE 35

1. Controller Processes 35

2. The User-Interface Process 37

3. Backend Processes 37

4. Communication Processes 38
C. THE COMMUNICATION PROTOCOLS 39
1. Interprocess Communications 40
2. Interprocessor Communications 43
IV. THE DISK INPUT/OUTPUT PROCESS 45
A. DESIGN CONSIDERATIONS 46
1. Design Objectives 47
2. Design Alternatives 48
3. Portability Considerations 49
B. THE DESIGN OF THE DISK I/O PROCESS 50
1. The Message Passing Interface 53
2. The Queuing Strategy 54
3. The I/O Processing 54
C. THE IMPLEMENTATION 56
1. The Program Structure 56
2. Modifications in Record Processing 58
V. CONCLUSIONS 60
A. A REVIEW OF THE MBDS SYSTEM 60
B. THE SUMMARY OF THE WORK 61
C. PORTABILITY ISSUES AND SOLUTIONS 62
APPENDIX A - THE USER INTERFACE 64
APPENDIX B - DISK I/O SPECIFICATIONS 82
LIST OF REFERENCES 91
INITIAL DISTRIBUTION LIST 92

I. AN INTRODUCTION

The Multi-Backend Database System (MBDS) [Ref. 1 and 2] is an on-going effort directed by Prof. David K. Hsiao of the Naval Postgraduate School for conducting research in large-scale database processing. The system consists of a controller and a varying number of parallel backends. The controller provides the required user interface and orchestrates the activities of the backends. Backends, on the other hand, are equipped with large memories and high-capacity disks to handle the bulk of the parallel database operations.

MDBS is designed to be a highly extensible system. Performance improvement can be achieved by adding new backends. By the same token, database growth can be accommodated without sacrificing performance. Because backends have identical hardware and are running identical software, additions of new backends require no software modification other than redistributing the database.

A prototype of such a system has been developed. This prototype uses a VAX-11/780 computer as the controller and two PDP-11/44 computers as backends. The controller and the backends are interconnected via a number of time-division multiplexed links called the Parallel Communication Links (PCLs). User requests are entered from a host computer into the controller which simultaneously broadcasts the requests to the backends over the PCLs. Each backend, in turn, processes the same requests with concurrency control, directory management and record processing techniques [Ref 3 thru 6].

A. THE INTENT OF THE THESIS

Our goal of the thesis is first to examine the portability of the MBDS software. We plan to download the MBDS software from the configuration of the VAX and PDP hardware and VMS and RSX operating systems to the configuration of the microprocessor-based ISI-workstations and UNIX operating systems. We need to identify and isolate the configuration-dependent portions of the MBDS software and to carry out the necessary modifications and re-instrumentations in order to make the downloading successful. Our ultimate goal is to recommend to the future database-system designers the way to minimize the amount of configuration-dependent software and to strive for the design of a highly-portable database system for a variety of configurations.

As we begin our task of porting the MBDS software, Prof. Douglas S. Kerr of the Ohio State University has made available a version of the MBDS system software that has been downloaded to the configuration of a VAX-11/780 and three SUN-workstations running under Berkeley 4.2 BSD UNIX operating systems. Prof Kerr, through a previous affiliation with Prof. Hsiao, is one the member of the MBDS research project. He is still collaborating the database systems research with members of the MBDS project at the Naval Postgraduate School. Thus, we begin our portability study with the Ohio State version. Because of the similarity between the SUN- and the ISI-workstation, our downloading effort is substantially reduced. The work involves (1) the downloading of the MBDS software to run on the ISI-workstation configuration under the UNIX operating system and (2) the design, development, and implementation of the new interface between the controller and the backends. There are three major areas of concern:

1) Multiple-record templates for the support of multiple file types in MBDS.

2) Intraprocess and interprocess communications based on the broadcast bus (i.e., the Ethernet).

3) Disk I/O operations for real-time access.

B. SOFTWARE PORTABILITY ISSUES

Software portability allows the software to be run on a second computer with lesser effort than it would be required to develop the software from the scratch for the second computer. The technique required for producing portable software varies with applications and their intended usages. For commercial software products, the questions of legal protection also become an important issue.

In general, software portability issues are divided in three categories: hardware dependencies, programming languages, and operating systems. The classic approach in dealing with hardware dependencies is first to identify portions of the software that may have to be changed for the different hardware so that the differences are modularized with well-defined interfaces, and then to design and develop these portion of the software for the new hardware. We believe that the software written in high-level languages does not alone guarantee the portability. Possible problems with the portability of the software written in high-level languages are differences in language implementations and dialects, as well as hardware dependencies that are inherent in the compilers. For coping with operating system idosyncrasies, the technique of the abstract construct is often used. Abstract constructs do not commit themselves to a particular operating system and computer hardware. They are therefore portable. If the next operating system differs from the original one, only the software module implementing the abstract constructs requires to be changed.

An ideal portable software system would be one that could be run on any computer with any operating system with no change at all. This is unlikely to happen. Nevertheless, it is important to note when designing portable software, the real issues are (1) to be aware of hardware dependencies rather than to avoid proper utilizations of hardware elements and high-level programming and (2) to utilize abstract constructs for operating system interfaces.

C. THE ORGANIZATION OF THE THESIS

The remainder of this thesis is organized in four chapters. Chapter II describes the implementation of multiple-record templates. Included in this chapter is a discussion on the data model and the data language. This discussion is necessary for the understanding of the concept of templates. The format of the record template is then discussed along with the loading of the database. Details of the database load is given in appendix A. It is hoped that this section together with the addendix is sufficiently well documented to serve as a MBDS user guide.

Chapter III discusses the Ohio State version of the message-passing interface. For the discussion of the message interface, the entire MBDS process structure is reviewed and shown how the messages are being handled. The handling of the message is a complex operation and may be difficult to understand. Therefore, it is necessary to introduce an overview of the Berkeley 4.2 BSD interprocess communication facilities, with an emphasis on the specific communication and broadcasting capabilities. In particular, we are interested in the messages issued by a program for the broadcasting network connections from within a program. However, the discussion of network application utilities is not included.

Chapter IV deals with the Disk I/O process. It is a new backend process to handle the disk I/O requests from other backend processes. This chapter

discusses the definition of the software requirements and the analysis of the design. Three design alternatives are presented and evaluated. Selection criteria are based on performance and portability considerations. Included in this chapter, are the design and the implementation. Detailed design specifications are given in appendix B. Finally, in chapter V, we summarize our portability efforts and recommend ways to improve the performance and portability of MBDS.

II. MULTIPLE-RECORD TEMPLATES

A record template (or template) is a specification of a record structure that the database administrator uses to characterize the organization of records in a file. We define a database to be a collection of files, a file to be a collection of records, and a record to be a collection of fields (or data items). Based on these definitions, we can describe the structure of a record in terms of the number of data items, the names (or attributes) of the individual data items and the associated data types and values. In doing so, we can separate the description of the record away from the actual records and keep the record description in a template. The template can later be used for determining and specifying the characteristics of a data item and its relation with other data items in a record. When records, so specified, are collected to form a file, the file structure would have the same attributes and similar relations among records in the same file. Because the structural information is maintained in a single template, a file structure can be reorganized by simply changing the template. Moreover, additional templates can be created to organize new files without inducing unnecessary redundancies or duplication of records. File reorganizations and multiple file organizations using multiple templates are often needed to reflect new applications and user requirements [Ref 7].

The previous MBDS implementation did not support multiple templates; only a single template was maintained. With only a single template, the notion of a database, as defined, is reduced to that of a file. Such a reduction on the database can have a severe impact on data manipulations. At a first glimpse, it seems

unimportant to differentiate between a database and a file since they both represent the same collection of records. Upon a careful consideration, however, we see that records in a file are of a given record type that is specified by a given template. Records in a database, on the other hand, are of multiple record types serving many applications. With a single template, however, a database is restricted to serve but one application.

To alleviate the limitation of a single application, the capability for multiple-record templates has been implemented in MBDS. This implementation extends the notion of the database and allows a database to have more than one file. For this implementation, it has been necessary to redesign the template structure and modify the existing template modules to implement the new template structure. Before we proceed with the discussion of the multiple templates, let us first review the data model and data language used in MBDS.

A. THE DATA MODEL AND DATA LANGUAGE

In this section, we will introduce the concept and terminology of the attribute-base data model which is the data model used in MBDS, and describe the data language for which users are required to issue requests.

1. The Attribute-Based Data Model

In the attribute-based data model, the data is considered in the following constructs - database, file, record, attribute-value pair, keyword, attribute-value range, directory keyword, non-directory keyword, directory, record body, keyword predicate, and query. Informally, a *database* consists of a collection of files. Each *file* contains a group of records which are characterized by a unique set of keywords. A *record* is composed of two parts. The first part is a collection of *attribute-value pairs* or *keywords*. An attribute-value pair is a member of the

Cartesian product of the attribute name and the value domain of the attribute. As an example, <POPULATION, 25000> is an attribute-value pair having 25000 as the value for the population attribute. A record contains at most one attribute-value pair for each attribute defined in the database. Certain attribute-value pairs of a record (or a file) are called the *directory keywords* of the record (file), because either the attribute-value pairs or their attribute-value ranges are kept in a *directory* for identifying the records (files). Those attribute-value pairs which are not kept in the directory are called *non-directory keywords*. The rest of the record is textual information, which is referred to as the *record body*. An example of a record is shown below.

(<FILE, USCensus>, <CITY, Monterey>, <POPULATION, 25000>,
{ Temperate climate })

The angle brackets, <,>, enclose an attribute-value pair, i.e., keyword. The curly brackets, {,}, include the record body. The first attribute-value pair of all records of a file, by convention, is the same. In particular, the attribute is FILE and the value is the file name. A record is enclosed in the parenthesis. For example, the above sample record is from the USCensus file.

The records of the database may be identified by keyword predicates. A *keyword predicate* is a 3-tuple consisting of a directory attribute, a relational operator (=, !=, >, <, >=, <=), and an attribute value, e.g., POPULATION >= 20000 is a keyword predicate. More specifically, it is a greater-than-or-equal-to predicate. Combining keyword predicates in disjunctive normal form characterizes a *query* of the database. The query

(FILE = USCensus and CITY = Monterey) or
(FILE = USCensus and CITY = San Jose)

will be satisfied by all records of the USCensus file with the CITY of either

Monterey or San Jose. For clarity. we also employ parentheses for bracketing conjunctions in a query.

2. The Attribute-Based Data Language (ABDL)

The attribute-based data language supports the five primary database operations, INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. A *request* in the ABDL is a primary operation with a qualification. A *qualification* is used to specify the part of the database that is to be operated on. Two or more requests may be grouped together to form a *transaction.* Now, let us illustrate the five types of requests and forgo their formal specifications.

The INSERT request is used to insert a new record into the database. The qualification of an INSERT request is a list of keywords with or without a record body being inserted. In the following example, an INSERT request that

INSERT (<FILE, USCensus>. <CITY. Cumberland>.
<POPULATION, 40000>)

will insert a record without a record body into the USCensus file for the city Cumberland with a population of 40,000.

A DELETE request is used to remove one or more records from the database. The qualification of a DELETE request is a query. The following example is a request that

DELETE ((FILE = USCensus) and (POPULATION > 100000))

will delete all records whose population is greater than 100,000 in the USCensus file.

An UPDATE request is used to modify records of the database. The qualification of an UPDATE request consists of two parts, the query and the modifier. The *query* specifies which records of the database are to be modified. The *modifier* specifies how the records being modified are to be updated. The

following example is an UPDATE request that

UPDATE (FILE = USCensus) (POPULATION = POPULATION + 5000)

will modify all records of the USCensus file by increasing all populations by 5,000. In this example, (FILE = USCensus) is the query and (POPULATION = POPULATION + 5000) is the modifier.

The RETRIEVE request is used to retrieve records of the database. The qualification of a retrieve request consists of a query, a target-list, and a by-clause. The query specifies which records are to be retrieved. The target-list consists of a list of output attributes. It may also consist of an aggregate operation, i. e., AVG, COUNT, SUM, MIN, MAX, on one or more output attribute values. The optional by-clause may be used to group records when an aggregate operation is specified. The RETRIEVE request in the following example will retrieve

RETRIEVE ((FILE = USCensus) and (POPULATION >= 50000))
(CITY, POPULATION)

the city names and populations of all records in the USCensus file whose populations are greater than or equal to 50,000. ((FILE = USCensus) and (POPULATION >= 50,000)) is the query and (POPULATION, CITY) is the target-list. There is no use of the by-clause or aggregation in this example.

Lastly, the RETRIEVE-COMMON request is used to merge two files by common attribute-values. Logically, the RETRIEVE-COMMON request can be considered as a transaction of two retrieve requests that are processed serially in the following general form.

RETRIEVE (query-1) (target-list-1)
COMMON (attribute-1, attribute-2)
RETRIEVE (query-2) (target-list-2)

The common attributes are attribute-1 (associated with the first retrieve request)

and attribute-2 (associated with the second retrieve request). In the following example, the RETRIEVE-COMMON request

```
RETRIEVE ((FILE = CanadaCensus) and (POPULATION >= 100000))
                                    (CITY)
COMMON (POPULATION, POPULATION)
RETRIEVE ((FILE = US Census) and (POPULATION >= 100000))
                                    (CITY)
```

will find all records in the Canada Census file with population greater than 100,000, find all records in the US Census file with population greater than 100,000, identify records of respective files whose population figures are common, and return the two city names whose cities have the same population figures. ABDL provides five seemingly simple database operations, which are nevertheless capable of supporting complex and comprehensive transactions. We are also note that Database files defined herein are therefore different from operating system files.

B. SPECIFICATIONS OF NEW TEMPLATES

The process of constructing a database is controlled by three operating system (input) files namely - the template file, the descriptor file and the record file. The template file is used to define the template structure. The descriptor file contains the directory attributes and their descriptor definitions. The record file contains the input records. These files are used by MBDS for the formulation of record clusters.

To illustrate how the input files are created, let us consider a purchasing system. This system consists of purchase-order, part, and supplier records. The relationships among the three types of records are described in the form of a schema as shown in figure 1. This schema is normalized and represented as tuples as shown in figure 2. In this representation, each tuple shows explicitly the names of

the (file) templates and the attributes of each template (file). The process of normalization accomplishes two important functions. 1) It captures the data relation from one file to another file. 2) It provides a form of representation that is suitable for template specification. Normalizations requires that certain attributes (therefore. attribute values) appear in more than one file. The order-# in the purchase-order file, for example, is repeated in the part file and is combined with part-# to form a unique identifier. Such duplication, however, does not necessary mean that the attribute value is redundantly stored because normalizations are concerned with logical structures rather than physical organizations.

1. Template Descriptions

When a database administrator wishes to create a new database, the administrator begins by preparing a template file. The template file contains the descriptions of the templates defined in a database. In general, there can be many databases in the MBDS system. The template descriptions for and the records in different databases must be separate and disjoint.

The format of the template file for a given database with n templates is described as follows:

Database name
Number of templates in the database
Template description for template #1
Template description for template #2
.
.
.
Template description for template #n

A typical template description with m attributes is given as follows:

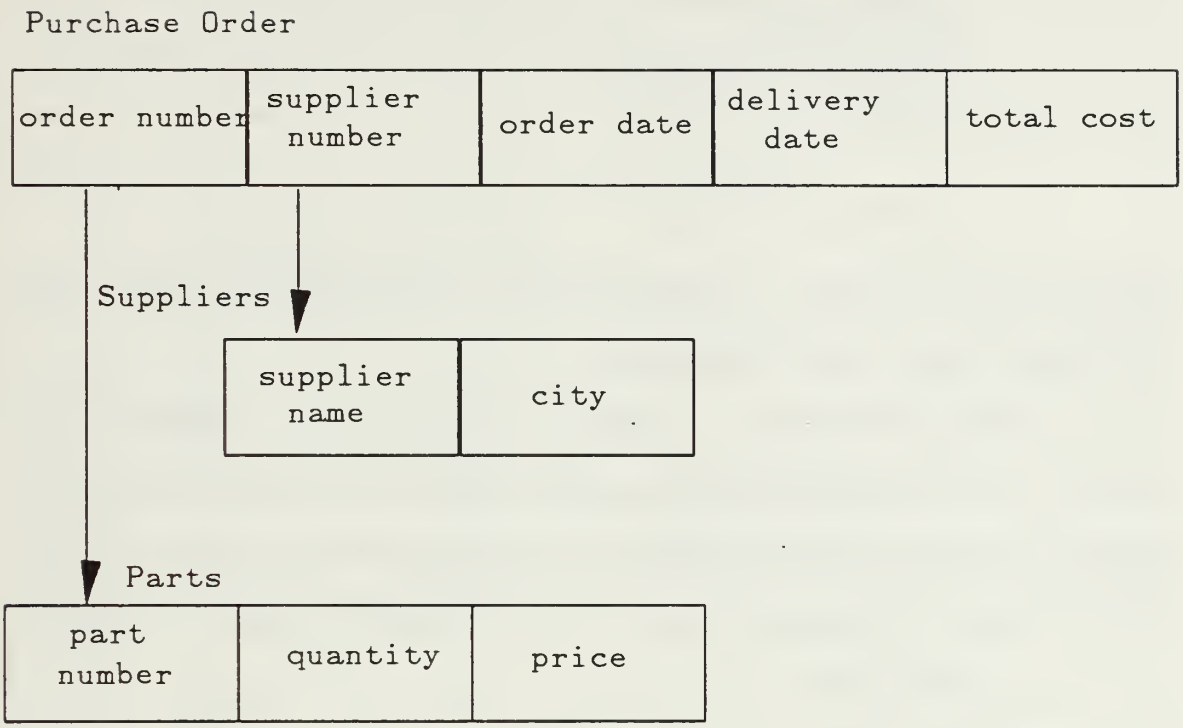

Figure 1 - Schema for a purchase order system

Purchase order (order number, supplier number, order date, delivery date, total cost)

Parts (order number, part number, quantity, price)

Supplier (supplier number, supplier name, city)

Figure 2 - Normalized form of purchase order system schema.

```
Number of attributes in a template
Template name
attribute #1  data type
attribute #1  data type
    .
    .
    .
attribute #m data type
```

There are three data types - integer, character string, and floating point. They are represented as i, s, and f respectively.

Having represented the purchasing database, in a normalized form, the creation of the template file becomes a simple matter of filling in the blanks. The name of the database is PURCHASING. The template names are Purchase-Order, Part and Supplier. Thus, the template file for PURCHASING is shown as follows:

```
PURCHASING
3
6
Purchase-order
template          s
order-#           s
supplier-#        s
order-date        s
delivery-date     s
total-cost        f
5
Part
template          s
order-#           s
part-#            s
quantity          i
price             f
4
Supplier
template          s
supplier-#        s
supplier-name     s
city              s
```

In a template description. the first attribute is called the system attribute. A system attribute is an attribute whose value is the name of a template. It is used to identify the records in a given template. Following the system attribute are the attributes of the corresponding tuples as shown in figure 2.

2. Descriptor Specifications

Our next job is to create a descriptor file. A descriptor is a keyword predicate of the form, for example, (supplier-name = DEC), (price >= $100), or (price < $10,000). MBDS recognizes two kinds of keywords: simple keywords (or non-directory keywords) for search and retrievals and directory keywords for formulating clusters. The descriptor file is an operating- system file that contains descriptors of directory keyword only. Cluster formulations are based on the attribute values and value ranges of the descriptors. For example, a cluster that contains the records in the purchasing database for purchase orders ordered from a supplier DEC with a total cost of $10,000 and up to $100,000 since June 1986, is derivable with a set of three descriptors - (supplier-name = DEC), ($10,000 =< total-cost < 10,000.00), and (order-date = June 1986).

There are three types of descriptors. A type-A descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate. An example of a type-A descriptor is (10,000 =< total-cost < 100,000). For creating a type-A descriptor, the database administrator needs to specify the attribute (i.e., total-cost) and the value range (i.e., of 10,000 and up to 500,000). The value range is expressed in term of upper and lower limits. A type-B descriptor is an equality predicate (i.e., supplier-name = DEC). A type-C descriptor is also an equality predicate. However, the values of the predicate are provided by the input records. Type-C descriptors are converted then automatically to a set of type-B descriptors with the same attribute name and values corresponding to the

value range. For example, if a template has a type-C descriptor with certain values of purchase-order, part, and supplier, provided by the records, then the first set of type-B descriptors generated are (template = purchase-order), (template = part), and (template = supplier).

The rule for specifying a descriptor requires that the attributes of the descriptors of a given be unique and that the values and value ranges in the specification be mutually exclusive. A format of a descriptor file that has n descriptors is given below:

```
Database name
Descriptor definition 1
Descriptor definition 2
     .
     .
     .
Descriptor definition n
$
```

The $ sign indicates the end of the descriptor file. Each descriptor definition in the descriptor file is expressed in terms of the attribute and its associated descriptor type and data type and followed by the value ranges as shown below.

```
Attribute  Descriptor-type  Data-type
Value range 1
Value range 2
  .
  .
  .
Value range k
@
```

The value range is expressed in terms of the lower and upper limits. For type-B and type-C descriptors, there is the exact value. The placehold for the upper limit is used for holding the exact value. The lower limit is not applicable. The value of the lower limit is therefore indicated by an ! character. The @ character signifies the end of the descriptor definition. For example,

```
PURCHASING
template            C                   s
!                   Purchase-order
!                   Part
!                   Supplier
@
total-cost          A                   f
1000.00             100000.00
100000.00           500000.00
@
order-#             A                   s
#1                  #50
#50                 #100
#100                #1000
@
price               A                   f
1000.00             50000.00
50000.00            500000.00
@
supplier-name       B                   s
!                   DEC
!                   IBM
!                   ISI
@
city                B                   s
!                   Monterey
!                   San Clara
!                   San Josa
@
$
```

3. Database Records

Once the template and the descriptor files are defined. data records can be specified accordingly. The record file can be prepared in separate files for subsequent loading. The format of a typical record file is given below.

Database name

@

Template name #1

Record #1 for template #1

Record #2 for template #1

Record #n for template #1

@

Template name #2

Record #1 for template #2

Record #2 for template #2

Record #n for template #2

$

The database name identifies the database to which the templates and the records belong. The @ sign signifies the beginning of a new template followed by the name of the template. All records below the template name belong to that template until another @ sign or a $ sign is encountered. The $ sign indicates the end of the entire record file. Each record contains the values of the attributes in the template. Each value in a record is separated by at least one space. For example, a record for the purchase-order template in the database PURCHASING has the following attribute-value pairs - (see template file section B.1 above):

<order-#, 26>, <supplier #, 51>, <order-date, 18 May 1981>,

<delivery-date, 22 Nov 1985>, <total-cost, $191,500.00>

Note that in the template file, the first attribute is the attribute name of a template whose value is purchase-order which can be represented also as a attribute-value pair (i.e., <template, purchase-order>). The template attribute is omitted from the record; instead, it is placed above the records. if we extract the values of each attribute-value pair, we have

26 51 18-May-1985 22-Nov-1985 $191,500.00

which is the first record of the purchase-order template in the purchase-order database. An example of a more complete record file as follows:

```
PURCHASING
@
Purchase-order
26    51      18-May-1985    22-Nov-1985    191500.00
31    51      25-Jun-1985    14-Apr-1986    381.900.00
@
Part
26    V780    VAX-11/780     1              91500.00
26    M780    Memory         8              42000.00
26    D81     RA81           3              58000.00
31    V8600   VAX-8600       1              381.900.00
@
Supplier
51    DEC     San Clara
$
```

C. LOADING THE DATABASE

The procedure for the loading of the database is composed of three phases - the database definition. the record handling. and the record clustering and placement. Phase I involves the specifications of the record template, the descriptor and the record files as discussed in the previous section. This section deals with the remaining two phases. Operational details are given in Appendix A.

1. Record Handling

In the MBDS implementation, the handling of input records is menu driven. During the system startup, an interface process is invoked for user interaction. This process begins by displaying the main menu and waits for a selection response by the user. The selection of an entry is made by typing the appropriate key as indicated and by following it with a return key. A scenario for loading the database is given below. For the purpose of annotation, each operation is numbered. The response for the appropriate action is given in the bold face immediately after the prompt > character. A single prompt character is given for each level of services.

The names of the input files described in the previous section are template.f. and descriptor.f and record1.f. If there are more than one record file, record files named record1.f, record2.f... can be loaded repeatedly at step 7. Step 8 begins the execution. This step and the subsequent steps are included to show how the database can be manipulated. The file request.f has been prepared for this purpose. It contains a list of transactions (known as traffic units) as shown in step 11. In step 11, a retrieval request is selected. The result of this transaction is displayed.

```
1 - What operation would you like to perform?
      (g) - generate database
      (l) - load database
      (e) - execute requests
      (x) - exit to operating system
      (z) - exit and stop MBDS

> l

2 - What operation would you like to perform?
      (t) - load template and descriptor files
      (r) - mass load record files
      (x) - return to previous menu

>> t

3 - Enter name of file containing template information:

>>> template

4 - Enter name of file containing the descriptors:

>>> descriptor.f

5 - What operation would you like to perform?
      (t) - load template and descriptor files
      (r) - mass load record files
      (x) - return to previous menu

>> r

6 - Enter name of file containing records to be loaded:

>>> record1.f

7 - What operation would you like to perform?
      (t) - load template and descriptor files
      (r) - mass load record files
      (x) - return to previous menu

>> x

8 - What operation would you like to perform?
      (g) - generate database
      (l) - load database
      (e) - execute requests
      (x) - exit to operating system
      (z) - exit and stop MBDS

> e

9 - Do you want to wait for responses? (y/n)
```

>> **y**

10 - Enter type of subsession you want:
(r) - REDIRECT OUTPUT: select output for answers
(d) - NEW DATABASE: choose a new database
(n) - NEW LIST; create a new list of traffic units
(m) - MODIFY; modify an existing list of traffic units
(s) - SELECT; select traffic units from existing list
(o) - OLD LIST; execute traffic unit in existing list
(p) - PERFORMANCE TESTING
(x) - EXIT; exit and return to main menu

>> s

11 - Enter name of traffic unit file:

>>>**request.f**

List of executable traffic units
(0) - [RETRIEVE(template=Part)(order-#,price)]
(1) - [INSERT(<template,Supplier>,<supplier-#,62>,<supplier-name,IBM>,
<city.San josa>)]
(2) - [DELETE((template=Purchase-order)and(oder-#=6225))]
(3) - [UPDATE((template=Part)and(oder-#=62))<price=49500.00>]
12 - Select options
(num) execute traffic unit number
(d) display traffic units in list
(n) enter a new traffic unit
(x) return to previous menu

>>>> **0**

order-# = 26
price = 91,500.00

order-# = 26
price = 42,000.00

order-# = 26
price = 48,000.00

order-# = 31
price = 381.900.00

13 - Select options
(num) execute traffic unit number
(d) dispaly traffic units in list
(n) enter a new traffic unit
(x) return to previous menu

>>>> **x**

14 - Enter type of subsession you want:

```
(r) - REDIRECT OUTPUT: select output for answers
(d) - NEW DATABASE: choose a new database
(n) - NEW LIST: create a new list of traffic units
(m) - MODIFY: modify an existing list of traffic units
(s) - SELECT: select traffic units from existing list
(o) - OLD LIST: execute traffic unit in existing list
(p) - PERFORMANCE TESTING
(x) - EXIT; exit and return to main menu

>> x

15 - What operation would you like to perform?
     (g) - generate database
     (l) - load database
     (e) - execute requests
     (x) - exit to operating system
     (z) - exit and stop MBDS

>.z.
```

2. The Record Clustering and Placement

To provide a better insight on the database loading process, we need to explain some aspects of the record clusters and placements. Formal discussions of these algorithms are given in Ref. 3. For the purpose of our discussion, let us consider a simpler version of the descriptor file and the record file of our purchasing example. The descriptors and the record, taken from the example, are rewritten in a more descriptive form below:

Purchase-order records:

(<order-#,#26>,<supplier-#,#51>,<total-cost,191.5K>)

(<order-#,#31>.<supplier-#,#51>,<total-cost,381.9K>)

Part records:

(<order-#,#26>,<supplier-#,#51>,<price,19..5K>)

(<order-#,#26>,<supplier-#,#51>,<price,42K>)

Supplier record:

(<supplier-#1>,<supplier-name,DEC>,<city,San Clara>)

Type-A descriptors and their descriptor ids:	
(1K =< total-cost < 100K)	D1
(100K =< total-cost < 500K)	D2
(#1 =< order-# < #50)	D3
(#50 =< order-# < #100)	D4
Type-B descriptor. its descriptor id:	
(supplier-name = DEC)	D5
A Type-C descriptor on templates only:	
(template = purchase-order)	D6
(template = part)	D7
(template = supplier)	D8

First, we convert the type-C descriptor into a set of type-B descriptors. For each type-A and each type-B, we assign a descriptor id. Thus, we have D1 through D8 as indicated above. Now we examine the records. The first purchase-order record has the keywords <order-#,#26>, <supplier-#,57>, and <total-cost,191.5K>. We see that the first keyword is in D3. The second keyword has no descriptor because supplier-# is not a directory keyword. It is used simply as a link to Part and Supplier records. We also see that the third keyword is in D2 and finally the template for Purchase-order records are in D6. By identifying the keywords of descriptor ids, we note that the record is in the cluster formed by the descriptor set {D2, D3, D6}. The following are the clusters and records of the clusters satisfying their descriptor-id sets.

Cluster 1	{D2, D3, D6}	records #1, #2
Cluster 2	{D1, D3, D7}	records #3, #4
Cluster 3	{D5, D8}	records #5

Clusters that are formed in this manner, are distributed across the backends for storage. Details of the cluster placement algorithm are discussed in Ref. 3. The same record clustering and placement technique can be used for search and retrieval of records in a cluster.

III. MESSAGE-PASSING FACILITIES

The MBDS software has been developed in stages and in versions; each subsequent version incorporating some updates. After incorporating the updates, the updated version becomes a new baseline. Updates within a baseline include the implementations of the planned activities and are often conducted in parallel. The earlier versions of MBDS began with a simple system which has finally developed to a complete prototype that runs on the VAX and PDP configuration [Ref 3-6]. The recent implementations of MBDS along with the significant updates in each version are as follows:

VerE.4	Prototype version - the controller software.
VerE.4a	Prototype version - the backend software.
VerE.4b	Ohio State implementation that runs on a single UNIX system.
VerE.4T	NPS implementation of 4b with multiple record templates.
VerE.4d	Ohio State implementation of 4b with multiple backends.
VerE.4e	NPS modification of 4d with broadcasting and templates.
VerE.4f	NPS implementation of 4e with the disk I/O.
VerE.4g	NPS implementation of 4e with the retrieve common primary operation [Ref 8].

The implementation of multiple record templates (VerE.4T) was first implemented on the VAX-11/750 running under the Berkeley UNIX 4.2 BSD. This has subsequently been updated into VerE.4e. VerE.4d is the Ohio State version of the MBDS software that runs on the VAX and SUN-workstations. In VerE.4e. The VerE.4d has been downloaded to run on the ISI-workstations. Because of the competability of the VAX. of the SUN and the ISI systems, the downloading effort reduced to changing the host names of the ISI-workstations so that the MBDS software could be recompiled and run on the ISI-workstations.

In this chapter, we describe the message-passing facilities of MBDS. This discussion is given in three parts. First, we discuss the ISI-workstation configuration, which are configured specifically to meet the MBDS requirements. These requirements include the hardware and the software supports for large-capacity disks, virtual memory, C-language programming, process-oriented operating environment, and interprocess communications. Next, we present an overview of the MBDS software structure and show functionally how messages are being handled in MBDS. Finally, we discuss the communication protocols. This discussion includes the message-passing mechanisms for communications between any two processes within the controller and between a controller process and a backend processor. The handling of the communication between the processes of workstations is different. The difference is to be discussed from points of view of the operating system and the MBDS software.

A. THE ISI-WORKSTATION CONFIGURATION

Figure 3 shows the MBDS hardware organization, of eight ISI-workstations with their disk subsystems. The ISI-workstations are interconnected via an Ethernet broadcast bus. ISI-8 is shown to be the controller with ISI-1 through 7 acting as backends. This configuration, however, is temporary due to the shortage of the Ethernet hardware interface boards and cables. The final configuration will have a VAX-11/750 serving as the controller and eight ISI-workstations as backends.

As shown in Figure 3, the user access is accomplished through a host computer to the controller. When a transaction (a single request or a sequence of requests) is received, the controller broadcasts the transaction to all the backends. Since the database is distributed across the backends, all backend processors execute the same request in parallel. Each backend maintains its own request queue.

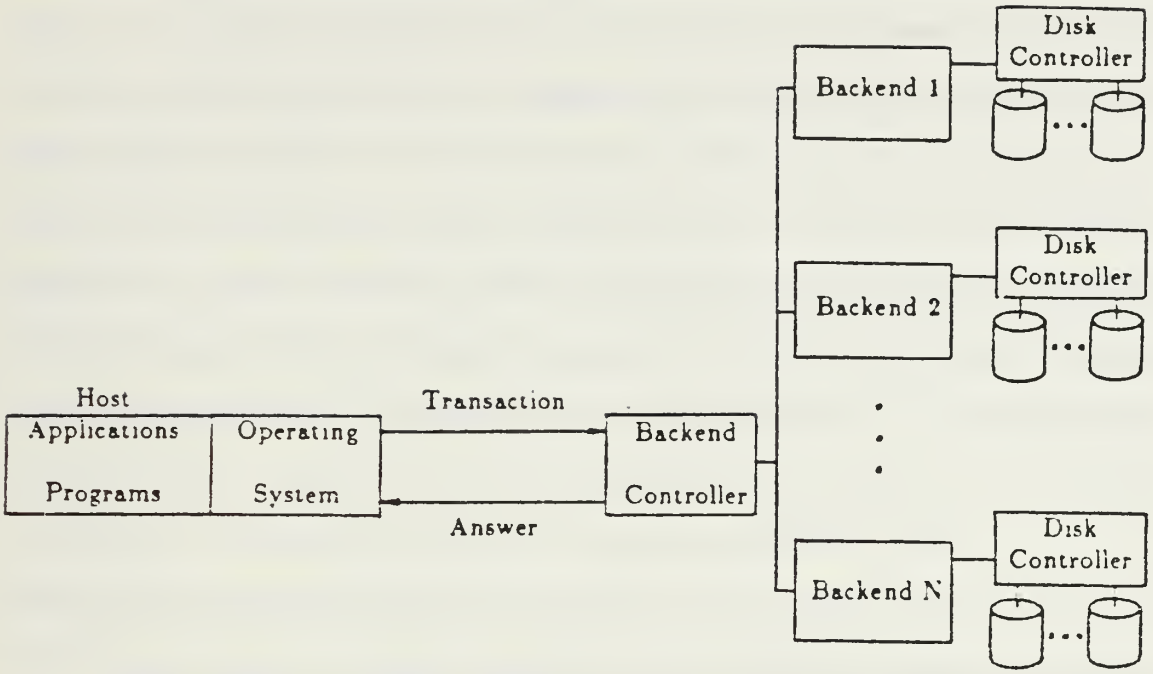

Figure 3. The Multi-Backend Database System.

As soon as a backend finishes a request, it sends the result back to the controller and continues to process the next request independent of the other backends.

1. The Hardware Organization

The ISI-workstations are microprocessor-based computers manufactured by Integrated Solutions, Inc., a NBI company. Each workstation is configured around a MC68020 processor on a VME bus, two-Mbyte CPU memory, a 106-Mbyte system disk drive, a 515-Mbyte database disk drive, and an Ethernet interface.

The Motorola MC68020 is a 32-bit processor with 32-bit registers and data paths, an internal instruction cache, and the pipeline instruction execution. This processor also supports a 256-Mbyte virtual address space and up to 16-Mbyte physical address space with demand paging. Translation from virtual

address to physical address is based on 4-Kbyte pages. The page table entries are kept in a high-speed buffer memory and maintain the most recently used translations. When a page is needed for execution, the hardware sequencer pulls the translation from the page table into the translation buffer and automatically updates both the access bit and the modified bit as required. Each page entry has additional protection bits indicating no access, read-only access, or read/write access. If a program attempts to execute a memory page that is not permitted by these protection bits, the hardware prevents the cycle from occurring and causes a bus error.

Each workstation has two CDC disk subsystems. A 106-Mbyte 5-1/2-inch Winchester drive is used to support the operating system and the MBDS software. The other 515-Mbyte 9-inch Winchester drive is used as the database store. The database drive has seven disks mounted on a spindle. The drive motor rotates the disks at 3600 rpm. The maximum seek time for a cylinder is 45 msec and the seek time for a single track is 5 msec. There are 25 heads: a servo head to control the actuator positioning, and 12 pairs of read-and-write heads - one for read, one for write - for data transfer to and from the disks. Each drive has 711 cylinders, 12 tracks per cylinder, and 60 Kbytes/track.

An Ethernet interface is also provided for each workstation for connection to an Ethernet cable via a transceiver to form a network which encompasses the VAX-11/750 and the eight ISI-workstations.

In addition, four of the ISI-workstations are equipped with graphics options which include the display memory and the graphics controller, a high-resolution monitor and a three-button mouse. Although the graphics support is intended to assist the MBDS development on the ISI-workstations, future MBDS implementation may well include graphics for database applications.

2. The Software Environment

The ISI-workstations are operating under the Berkeley UNIX 4.2 BSD with enchancements made to support grahpics, real-time, and networking applications. The Berkeley UNIX 4.2 BSD is the Berkeley version of the UNIX operating system designed to support the VAX family of computers. This version includes the extended network and the interprocess communication facilities and many new features. For the MBDS development, we have found this environment to be especially desirable, because the UNIX-based systems are portable and run on a wide range of computers from microprocessors to large mainframes.

B. THE MBDS PROCESS STRUCTURE

Figure 4 shows the MBDS process structure. MBDS, as a message-oriented system, is composed of independent processes. The processes are designed to support the database functions of the controller, these processes are the request preparation, insert information generation, and post processing. Also running in the controller, is the user interface process. For the backends, these processes are the directory management, record processing and concurrency control. Each of the backends run identical processes. In addition, common to the controller and the backends are two communication processes. All MBDS processes are created at the system start-up time and run throughout the MBDS session.

1. Controller Processes

The request preparation process receives a request from either the user interface processor or the host computer, parses the request and checks it for syntax. The request is then classified according to its type and broadcasted to the backends. For a retrieve request with an aggregate operator such as sum, average, max or min, the aggregate operator is also sent to the post processing process

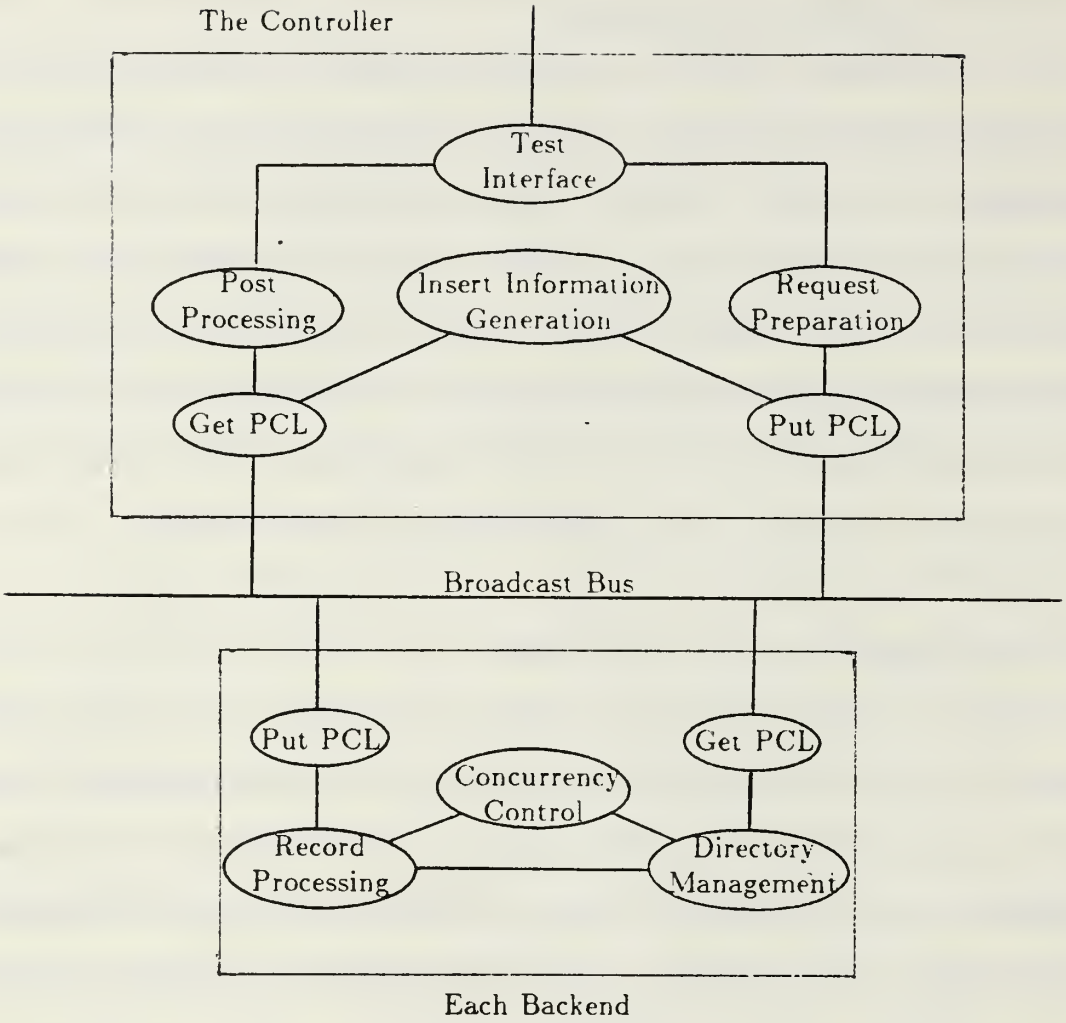

Figure 4. The MBDS Process Structure.

before being broadcasted to the backends. In the case of an update request, the request preparation process generates a sequence of requests to the backends, so that the old records can be marked for deletion and that the new records can be inserted.

The insert information process supplies the additional information requested by a backend while processing an insert request. This information, which can only be determined by the controller, includes the cluster id of the

cluster to which the records to be inserted may belong. The insert information is returned to the backend selected by the placement algorithm, and not necessarily to the requesting backend. If the record to be inserted involves new descriptors the new descriptor and the identification number are broadcasted to all the backends.

The post processing process collects the results from each of the backends, performs any additional processing as required, and delivers the final result to the requesting host or the user interface process. Should any aggregate operators be received from the request preparation process, aggregate operations will be performed prior to the delivery.

2. The User Interface Process

For the direct user interaction without going through the host, the controller, provides a user interface process to control the individual MBDS sessions. A MBDS session is established with a session request from the user. If the session is granted, a user interface process is created to handle the transactions and it is terminated when the session is finished. The user interface process begins by displaying a main menu and then waits for a selection response from the user. A selection response by the user is made by typing the appropriate key, and by following it with the return key. The options provided to the user include the loading of the database from the existing files, generating a new database and executing the user requests. A detail description of the database operations is given in appendix A.

3. Backend Processes

The directory management process, as a backend process, receives requests from the controller. Its major functions are the directory search, the directory maintenance, and the determination of the secondary-storage record

addresses. For managing the database, the directory management process maintains a set of directory tables: namely, the attribute table, the descriptor table, and the cluster definition table. The attribute table contains attributes of the database which identify the characteristics of the database. This also links to the descriptor table. The descriptor table consists of a set of predicates which define the values of the attributes and the corresponding descriptor-ids. As discussed in chapter 2, there are three types of predicates: the predicates that define a range of attribute values, the equality predicates, and the predicates that define a subset of all unique attribute values in the database. Finally, the cluster definition table contains cluster-ids and descriptor-id sets whose descriptors define a cluster, and addresses of the records in the cluster.

The record processing performs the disk access operations. The currency control process allows concurrent accesses of both directory data and base data. Locking mechanisms are provided to ensure their consistency and integrity. Aside from the four basic requests: insertion, deletion, retrieval, and update, there are many more operations derived from the combinations of these basic requests. Interdependencies among the directory management process, record processing process and concurrency control processes are discussed in detail in references 3-6.

4. Communication Processes

The communication between the computers is achieved via the Ethernet. MBDS provides a software abstraction to the Ethernet for both the controller and the backends. The abstraction consists of two complimentary processes. The first process, get-net, gets the messages from the other backends and the controller off the Ethernet. The second process, put-net, is used to send messages to the other backends or to the controller or to broadcast the messages

to all of them. Every computer, whether it is the controller or a backend, has its own get-net and put-net processes. In addition to the processes described in previous sections, these communication processes add up to a total of six processes for the controller and five for each backend.

When it gets a message, the controller get-net process first determines the message type, and then re-routes the message to the appropriate controller process. On the other hand, the controller put-net process receives a message from the other controller processes and broadcasts the message to all the backends. For the backends, the backend get-net process gets a message from either the controller or other backends, examines its message type, and dispatches the message to the appropriate backend processes. The backend put-net process receives a message from the other backend processes. Depending on the message type, messages are either broadcasted to all other backends or sent directly to the controller.

C. COMMUNICATION PROTOCOLS

To support the MBDS message-passing abstraction, there are three software modules to handle specifically three types of communications: namely, interprocess communication (that is the communication of processes within the same computer), interprocessor communication (that is the communication of processes in difference computers) and broadcasting (that is the communication among processes in one computer to all other computers). These modules are catered to the Berkeley UNIX 4.2 BSD implementation. Interprocess and interprocessor communications under the 4.2 BSD are established via sockets. Each socket is characterized by a socket type and a communication domain. The 4.2 BSD supports three socket types and two communication domains. The socket

types are used to define the communication protocols. A stream socket supports the transmission control protocol and the internet protocol commonly known as TCP/IP which are the DARPA standard communication protocols. A datagram socket supports the user datagram protocol and the internet protocol referred to as UDP/IP which are used for Ethernet communications. The UDP/IP protocols are not promised to be reliable in that the positive acknowledgement, timeout, and retransmission are not provided. Finally, a raw sockets support protocols to suit your own requirements. It is used for the development of non-standard networks. The two communication domains supported under the 4.2 BSD are the UNIX domain and the Internet domain. To put it simply, the UNIX domain is used for interprocess communications and the Internet domain is used for interprocessor communications.

1. Interprocess Communications

For the communication between the processes within the controller or a backend, a datagram socket in the UNIX domain is used. At the system startup time, each process creates its own socket and connects the socket to the other sockets which communications are required. Once the socket connection is established, sockets can exchange messages by using the send and receive procedure calls.

The communication of two processes are based on a client/server model. The action required to establish the communications is asymmetric. The client process is required to establish a connection to the server's socket which is known. The server must accept the connection so that the client's socket can be made known. The server process begins by creating a socket with his own well-known socket address. It assigns a name to the socket by a binding process and listens for a connection from a client. When the connection from a client is made,

the server process may accept the connection. If the server accepts the connection. communication between the client and the server is established. or else the server will wait. Meanwhile, a client may request service from the server by connecting to the server socket. The processes of creating a socket, binding it, listening for a connection. and accepting a connection as well as making a connection, are system calls. This procedure can be summarized as follows.

On the server side:

```
my-socket = create( domain, socket-type):
bind( my-socket, server-name);
listen( my-socket);
while forever
    {
    his-socket = accept( my-socket);
    if( his-socket is connected)
        save his-socket for future communication
        and return
    }
```

On the client side:

```
my-socket = create( domain, socket-type);
bind( my-socket, client-name);
connect( my-socket, server-socket);
```

To establish the communication sockets in MBDS, a common routine is provided. This routine is table driven. There are two separate sets of tables, one for the controller and other is common to all the backends. The tables are the process-name table, the connection table and the socket-name table. The

process-name table contains the name of the processes represented as an array. It is used to determine the name of the calling process. The connection table is a square matrix whose order corresponds to the size of the process-name table. This table is used to determine whether the calling process is acting as a server or a client so that proper action can be taken to establish the required connection. Finally, the socket-name table is used to save the name of the established sockets for future reference. For example, A backend process-name table is given as follows:

concurrent control
backend put-net
backend get-net
record processing
directory management

The backend connection table is also given as follows:

n, n, c, c, c
n, n, c, c, c
a, a, n, c, c
a, a, a, n, c
a, a, a, a, n

The element (i,j) represents the action required by the i-th process in order to establish communication with the j-th process. The action elements are defined as follow:

n - no action required; c - requires a connection to the destination socket; a - requires the acceptance of the connected socket

If the calling process is the concurrent control process, it can be seen from the first row of the connection table that the concurrent control process is a client process

to the backend get-net process. the record processing process and the directory management process. By examining the socket-name table of the respective processes. the concurrent control process will be able to establish the required connections.

2. Interprocessor Communications

In this section, we present the interprocessor communication facilities in two areas: 1) the communication of two processors, and 2) the broadcasting. In general. the datagram sockets in the Internet domain are used for the interprocess communication in a distributed environment. The process of establishing socket connections discussed in the previous section is the same except that the socket address and the message handling are different.

In the Internet domain. a socket name contains an internet address and a port number. For MBDS, the Internet address is obtainable using the gethostbyname library call and the port number is pre-assigned. Unlike the interprocess communication facility where processes are communicating directly to one another. the interprocessor facility routes the messages through get-net and put-net processes. For example. process a in processor A wishes to communicate with process b in processor B. Process a must first sends the message to the put-net process in processor A. The put-net process. in turn, routes the message to the get-net process in processor B. In processor B. the get-net process dispatches the message to process b.

In the previous version of MBDS. the broadcast feature was simulated. The simulated broadcast approach is time-consuming and requires the same message be sent repeatedly to every processor in the network. More recently, the true broadcast feature has been implemented. The implementation itself is relatively simple. We have discovered that when a message is broadcasted,

the message is sent simultaneously to every host in the network including the sender and may cause the message to be lost. To correct the problem, we first record the name of the broadcasting processor. When the broadcast message is received, the message is discarded by the broadcasting processor.

IV. THE DISK INPUT/OUTPUT PROCESS

The disk input/output (disk I/O) process is a new process of the MBDS backends that stores and retrieves data on disks. In the early prototype implementation of MBDS, the disk I/O function has been an integral part of the record processing process in the backends. In addition to performing operations such as record selections, attribute-value extractions, and aggregate functions, the record processing process is also burdened with the handling of the physical disk I/O operations. Because the prototype is set out to prove the validity of the multi-backend design where relative performance has been the major concern, the absolute performance has not been of major concern. Thus, efficient disk I/O handling routines has been overlooked. But now, having proved the design concepts of MBDS and having established the performance gains achievable through the multiplicity of backends, we consider methods to improve the I/O efficiency.

In the present record processing process, the disk I/O operations are simulated as ordinary files. As ordinary files, the operating system imposes many layers of software, access methods, and buffers which MBDS has already provided for. Such duplications of software have placed unnecessary constraints on the size of the records and limited the capability for real-time access.

In this chapter, we present the design and the implementation of a disk I/O process. The presentation is given in three sections. First, we discuss the design considerations that include the requirements, the alternatives and the portability issues. Second, we discuss the design of the disk I/O process. Included in this discussion are the message passing interface, the message queuing strategy, and the

I/O processing algorithms. Finally, we discuss the implementation and the integration of the disk I/O process as an added process in the backend.

A. DESIGN CONSIDERATIONS

In the early prototype MBDS implementation, the input to the record processing process comes from the directory management process. The input is in the form of a request indicating the type of the request and a set of addresses where the relevant data can be found. Upon receiving the request, the record processing process allocates the required buffers and processes the request according to the request type. Associated with every request, there are two sets of buffers: a track buffer which contains the relevant data read from a disk as specified by the set of addresses in the input, and a result buffer which contains records that satisfy the request after being processed. Each set of buffers is self-contained, i.e., it is complete with request identifications, addresses, and status. For processing of the records, the track buffer provides the input and the records satisfying the request are placed in the result buffer. When the operation is completed, the results from the result buffer are returned to the controller for post-processing. In the case of an update request, the result buffer is sent back to the appropriate backends to be written on the disk.

Although the technique of buffering provides a high level of data abstraction and allows requests to be processed concurrently and independently, the degree of concurrency and independency is hampered by having to wait for the completion of the physical I/O, which is operating system dependent. The I/O operations under the Berkeley UNIX 4.2 BSD are synchronous, that is, when an I/O request is initiated via a call to the operating-system read/write routine, the return to the system call is blocked until the I/O operation is completed. If the I/O operation

requires a large block of data. which is generally the case in MBDS. an excessive among of waiting time is wasted for the I/O completion. Thus. one of the most important requirements for the design of the disk I/O process is that the disk I/O operations be asynchronous with respect to the record processing process, so that the record processing process may continue to process requests while the disk I/O operations are being completed. We now examine our design objectives for the disk I/O process. the design alternatives that we have considered and the portability issues that have guided our design.

1. Design Objectives

The primary objective for the design of the disk I/O process is to provide effective disk I/O operations. For effective disk I/O operations, we must first overcome the deficiencies that existed in the previous implementation and then device new algorithms for accessing the database disks. As discussed in the previous section, the disk I/O operations must be made asynchronous in order to allow the record processing process to operate independent of any pending disk I/O. Since I/O operations are synchronous under our current version of the operating system, we must find ways so that disk I/O operations become asynchronous. We must remove layers of unnecessary software and duplicated buffering so that the disk I/O can be performed in real-time. We must provide new algorithms so that I/O operations can be done continuously and smoothly without excessive seek time. Finally. we must also provide the support for multiple blocks of data on a single request without having to repeat the same request several times for several blocks.

The objectives of the design of our disk I/O process for effective I/O disk I/O operations can be summarized, therefore, as follows:

(1) the disk I/O operations be asynchronous with respect to the record processing process,

(2) the ability to access the disk in real-time,

(3) the provision for fast and efficient disk access strategies,

(4) the ability to read one or more blocks of data on a single request,

(5) the uniformity of message-passing interface.

2. Design Alternatives

With the above design objectives, we consider three different design alternatives for the design of the disk I/O process. They are:

(1) an intelligent driver approach,

(2) the forking of processes for disk I/O, and

(3) an added backend process.

The intelligent driver approach involves the development of a device driver tailored for MBDS operations. It includes the support for an efficient disk access algorithm and functions such as record selections and attribute-value extractions so that with a single call to the drive, records which satisfy a query can be retrieved or updated. As a driver, I/O operations can be made asynchronous. Furthermore, features such as real-time access, interrupts and error recovery can be facilitated. However, device drivers must be written and install for a particular operating system. For different operating systems, the requirements for developing a driver are different. Aside from the portability consideration, this approach represents a radical departure of the MBDS design. The design of MBDS software is based on well-founded engineering principles and requires the MBDS software to be easily written and maintained. For these reasons, the intelligent driver approach is, thereby, rejected.

The second alternative is the forking of a processes for the disk I/O. Forking is commonly used in UNIX to create a new process via the fork (system) call. When a fork call is executed, the calling process is split into two executing processes in a parent/child relationship. The child process runs in the same environment as the parent. Communications between the parent process and the child process is possible via an interprocess channel called a pipe. This mechanism can be used by the record processing process to create a child process to handle the disk I/O operations. The major drawback in this approach is that a fork call is required for every disk I/O. As a result, the amount of overhead for creating the child processes can be excessive. Because of this, the alternative of forking processes to handle the disk I/O is no longer being considered.

The third and final alternative, which has been adapted, is the addition of a disk I/O process on the backend processor. The disk I/O process becomes the sixth backend process in MBDS. The disk I/O process has two functions. The first function is to read a block of from a disk and the second function is to write a block of data to the disk. When the record processing process needs to read or write data on disk, it sends a message to the disk I/O process to perform the I/O operation and continues with other processing tasks. A complete design of the disk I/O process is discussed in the next section.

3. Portability Considerations

From the portability standpoint, the I/O operations is inherently dependent on the hardware and the operating system. This is because different devices require different device drivers and device drivers are an integral part of the operating system. For a database system, where the disk I/O plays a key role in storing and retrieving information on the storage device, there is no way that the hardware and operating-system dependencies can be totally avoided. The best

that we can do is to develop techniques to minimize the amount of hardware and operating-system dependencies. As discussed in chapter I. the technique of abstraction and isolation is often used for this purpose.

For the handling of the disk I/O in the disk I/O process, we support the use of a raw I/O device. The raw I/O is a mechanism provided by the operating system to transfer information directly between the user buffer and the device without the use of system buffers and (2) in a block size as large as the calling process wants to request. This approach involves setting up a special characters-oriented device called the raw device and uses the standard seek, read, and write system calls for subsequent accesses.

To minimize the amount of dependencies, we develop two high-level routines: one to setup the raw device (e.g., open(disk)). the other to read from or write to the disk (e.g., do-disk-io(read, data, here) or do-disk-io(write, data, there)). The disk I/O process which is independent of the hardware and the operating system. calls these high-level routines to perform the required operations. The routines are never-the-less operating-system dependent (e.g., seek, read, or write system calls). They have been localized and can be rewritten for other operating systems.

B. THE DESIGN OF THE DISK I/O PROCESS

Figure 5 shows a functional block diagram of the disk I/O process. The process begins at block 1 to perform the initialization function. This function includes establishing the communications with other backend processes, setting up the disks for the raw I/O operations, and initializing the data buffers and the I/O queues. After initialization, a routine to receive a message is entered at block 2. The message-receiving routine receives a message sent by the other backend

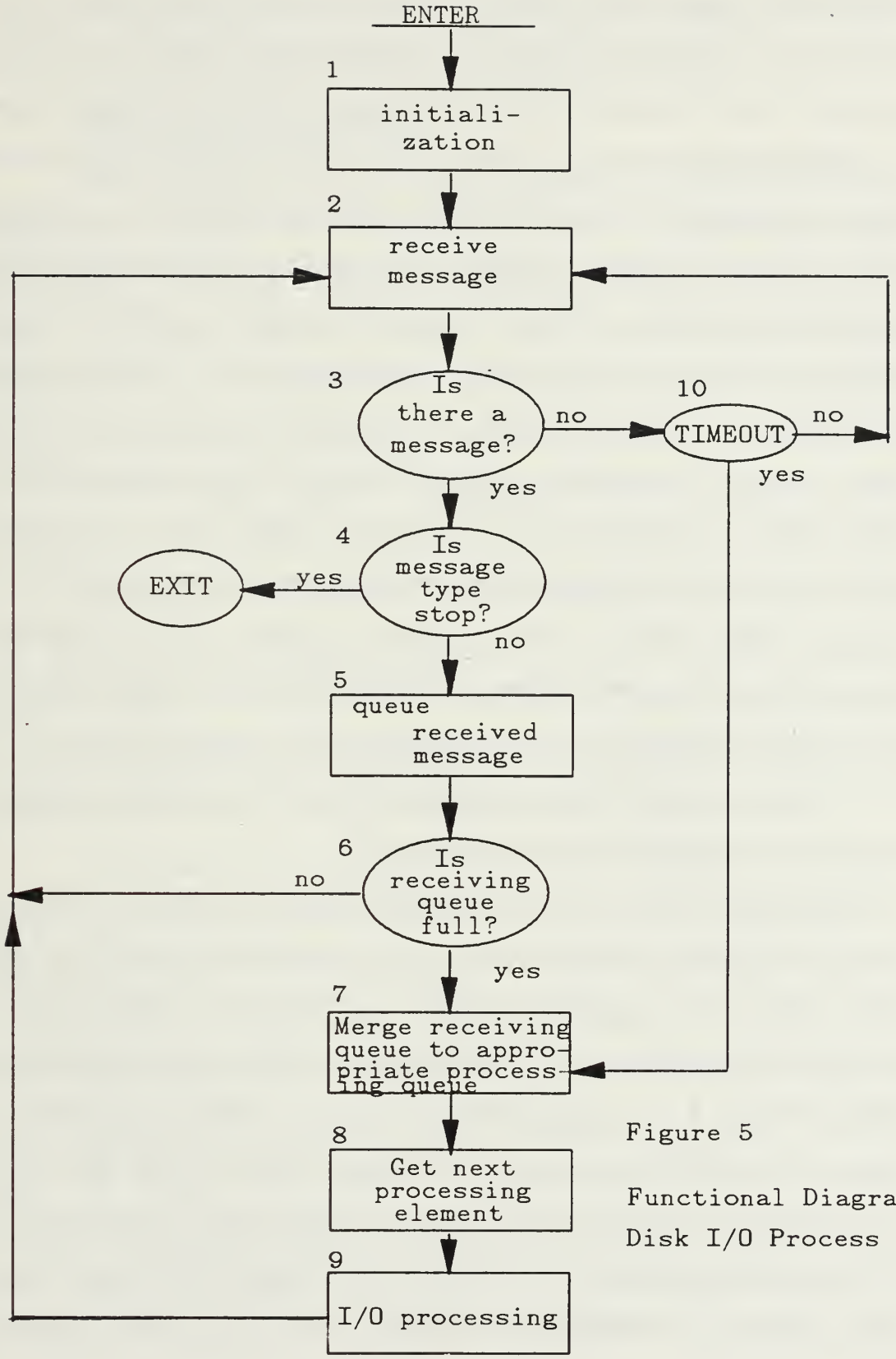

Figure 5

Functional Diagram
Disk I/O Process

processes and returns to the caller an indicator that indicating whether or not a message has actually been received. If a message has been received and the message type is "stop", the disk I/O process terminates; otherwise, a routine to queue the received message is entered at block 5. The received-message-queuing routine parses the message into a disk I/O queue element in the form of a quintuple (Sender, Request-id, Address, Operation, Block-pointer) and maintains a receiving queue of such quintuples. The disk I/O process continues to receive and queue additional messages as long as the receiving queue is not full. When the receiving queue is full, the entire receiving queue is merged into the appropriate processing queues. To merge the elements of the receiving queue with the appropriate processing queues. the elements are first separated by disks (if there are more than one disk connected to the backend processor) addresses. For each active disk, the get-next-processing-element routine is entered to choose a queue element from each processing queue that satisfies the chosen disk access strategy. After the queue element is selected, the I/O processing routine is entered to carry out the I/O operation as specified. Upon completion of the I/O operation. a message is returned to the requesting process with the result.

As indicated in figure 5, the entire program consists of three loops. The inner loop that encompasses blocks 2,3, and 10, is the message-wait loop. This loop is entered when there is no pending messages. It incorporates a timeout feature so that process can continue when the timeout occurs. The center loop encompasses blocks 2 through 6 and is the message-queuing loop. This loop is designed to queue up the messages as received until the queue is full. Once the queue is full, the queue may be merged and sorted to suit the required disk access strategy. The outer loop is the processing loop. This loop encompasses almost the entire program. After an I/O operation is performed for each active disk, the center

message-queuing loop is re-entered. If there are no further messages, the message-wait loop will timeout again and the I/O processing will continue.

1. The Message-Passing Interface

Since the sole function of the disk I/O process is to handle the disk I/O, communications are needed only with those processes that require disk accesses. At the present time, that process is the record processing process. Should the directory management process be required, in the future, to store and retrieve directory data on disks, the disk I/O process can easily be extended to provide the same service. Recall that in chapter III, we have discussed the interprocess communication facility which is used for communication between processes in a backend. Recall also, that communication sockets are established at the MBDS startup time and subsequent communication is done using high-level send and receive calls. When the record processing process needs to store or retrieve a record, a message is send to the disk I/O process. The disk I/O process, in turn, performs the required operation and send a message back. The information that pass back and forth between the two processes is contained in the message format.

A message consists of a message header and a message body. The message header identifies the sender, the intended receiver, and the message type. The message body contains the message itself. The message type indicates the type of the request. There are three types of requests: 1) a request to read one or more blocks of data from a disk, 2) a request to write a block of data onto a disk, and 3) the stop message which requests the disk I/O process to terminate. For a read request, the message body contains the request-id and one or more addresses where the relevant data can be found. For a write request, the message body contains a request-id, an address where the data is to be written followed by the data. Both the request-id and the address are required for the identification of the

track and the result buffers as discussed in section A. When the I/O operation has been completed, the request-id and the address field must be return in their original form along with the results so that the record processing process can identify the proper buffers which has been allocated prior to the request for disk I/O.

2. The Queuing Strategy

To enable the implementation of a disk access algorithm. the disk I/O process maintains one of each disk, a receiving queue and a set of processing queues, one for each disk. When a message is received, the message is parsed into a disk I/O queue element. The queue element is represented as a quintuple (Sender, Request-id, Address, Operation, Block-pointer). Each quintuple is associated with a single address. For a read request with more than one address, the request is parsed into a sequence of quintuples, as many quintuples as there are addresses.

The receiving queue is a list of quintuples. When the receiving queue is full, the entire receiving queue is merged into the appropriate processing queues. After each mergence, the receiving queue becomes empty. To ensure the processing of the I/O requests, the merging of the receiving queue can be performed regardless of the condition of the receiving queue. It is done by mean of the timeout feature of the message-wait loop. When the timeout occurs, the current status of the receiving queue is merged and become eligible for I/O processing. The timeout feature also ensure that no message in the processing queue remains unprocessed.

3. The I/O Processing

The processing of I/O is based on a disk access strategy. For this version, we have implemented a simple scheme to choose a queue element whose physical address lies in the direction of the access arm. Once the queue element is selected,

the queue element is removed from the processing queue. The required I/O operation is based on the information in the selected queue element. Recall that the queue element is a quintuple of the form

(Sender, Request-id, Address, Operation, Block-pointer).

The address field contains the disk number, the cylinder number and the track number. The disk number is used to specify the desired disk drive. If the backend processor has more than one drive, a separate process queue is provided for each drive so that all the drives can be accessed concurrently. The cylinder and track number are logical addresses suppled by the controller. They are used for deriving the physical track number of the disk drives where the operation is to be performed.

The operation field contains the desired operation. There are three types of operations: a single-address read, a multiple-address read, and a write operation. The difference of a single-address and a multiple-address read is in the processing, not the I/O operation. Recall that the multiple address is transformed into multiple single-address queue elements. The I/O operation is always done one track at a time. For the multiple addresses, the I/O completion message to be returned to the requesting process is deferred until the entire sequence is done.

The significance of the block-pointer is interpreted as follow:

1) For a write operation, the block pointer points to the data which is to be written onto a disk. When the write operation is completed, a write-done message will be returned to the requesting process.

2) For a read request with a single address, the block pointer is a null ponter. When the read operation is completed, a read-done message will be returned along with the data.

3) For a read request with multiple addresses, the block pointer is a pointer to an integer whose value represents the total number of addresses in the sequence. As each operation in the sequence is being processed, the value pointing to by the

block pointer is decremented. When the value reaches zero. the read request is completed. At which time. a read-done message will be returned along with the data for the entire sequence.

To complete our discussion on the queue element. The first field identifies the the sender of the request, so that the I/O completion message can be returned. The second field is the request-id which identifies the transaction.

C. THE IMPLEMENTATION

This section discusses the implementation of the disk I/O process. It is given in two parts. Part 1 presents the program structure and part 2 discusses the changes in the record processing process.

1. The Program Structure

The disk I/O program is organized in five basic modules. Each module is responsible for a particular function. They are the message handling module, the parsing module, the processing module, the entry selection module and the I/O module. The message handling module is responsible for receiving and sending messages. The parsing module transforms the message element into one or more I/O elements. Each element represents a pointer to the dio-info structure which contains the information needed to pass from one module to another. When a message is received, storages for the dio-info structure and data buffers are dynamically allocated. As the information is extracted from the message, the information is stored in the dio-info structure. The content of the dio-info structure is given as follows:

the sender-id

the space for the request-id structure

the space for disk address structure

the operation code

a ponter to a data block.

the number of data blocks for the request.

a link to the next dio-info structure

that contains the next data block.

a link head for the starting data block.

a link to the next dio-info structure

The first six fields are used for the I/O operations as discussed in the previous section. The next two links are used to support a single request that requires more than one block of data. For a multiple-block read request. the parsing module creates a dio-info structure and a block buffer for each data block requested. Since the data blocks may not be processed in sequence. they are linked within the dio-info structure for countabilities. The parsing module also maintain a list of dio-info structures using the last field. The same field is also used for the processing module to form its own processing list. Because the dio-info structures are bounded to a single operation, they are exclusive. The processing list consist of separate lists, one for each disk and sorted in ascending track address order. When the entry selection module is called, it selects one entry from each list according to the disk access algorithm and pass the list of entry to the I/O module for disk I/O operation. For a write operation. a returned message is sent to the requesting process via the message handling module. For multiple-block read request, the internal buffer chain is examined. A returned message is sent when all the blocks are read.

2. Modifications In Record Processing

The changes in the recording processing process are made in three areas: the removal of the disk initialization routines. the modifications for the message handle routine, and the changes for the I/O routines.

1) For the disk initialization,
   a) remove the sys-disk-init call io disk-init from disk-c
   b) remove the sys-disk-init routine in Unix disk.c

2) For receiving messages from the disk I/O process,

   a) insert in the main program in repro.c the following code:

```
case DIO:
    RP-DIO ();
    break;
```

   b) insert RP-DIO routine below RP-RP in recpro.c

```
RP-DIO ()
{
   get message type
   switch (message type)
      {
      case:  PIO-WRITE:
             RP-Write Completed);
             break:
      case:  PIO-READ:
             RP-Read Completed);
       }
 }
```

   c) modify RP-Read-Completed routine () in recpro.c to restore data from message buffer.

   d) delete case PIO-WRITE and case PIO-READ codes from RP-PR monitor io recpro.c

3) For sending message request to the disk I/O process,

   a) replace do-io call in TB=STORE and TB-FETCH routine in disk.c with the following code:

```
Reg-RP-S () where

Reg-RP-S ()
{
```

```
        fill message buffer with request-id
        fill message buffer with disk address
        in case of a write operation. fill
        message buffer with data
        set end of message
        header.receiver = DIO;
        set appropriate message type
        send (msg-q, &header);
    }
```

C

4) Add following code to RP_CNTL_ANOTHER_BE_MSG in recpproc.c to send stop message to DIO.

```
case (Stop):
    StopSys = TRUE;
    header.sender = RECP;
    header.receiver = DIO;
    send (msg_q, &header);
    break;
```

V. CONCLUSION

A. A REVIEW OF THE MBDS SYSTEM

The multi-backend database system (MBDS) consists of two or more processors and their dedicated disk systems. One of the processor serves as a controller to provide the host, user, and language interfaces while the rest of the processors serve as backends to provide the primary database operations with their disks serving as the database stores. All the backends are identical and run identical software. The database is evenly distributed across the disk drives of the individual backends using of a cluster-based placement algorithm. User sessions with the MBDS can be established either via a host computer, which in turn communicates with the controller, or with the MBDS controller directly. Communications of the controller and the backends are broadcasting-based with internal protocols and database structures.

MBDS, as a message-oriented system, is composed of independent processes. These processes are designed to support the communications and database functions. For the controller, there are six processes: two communication processes to provide a communication abstraction for the Ethernet, a user interface process to control the MBDS sessions, and three other processes to handle the user transactions. For each of the backends, there are six processes: two communication processes similar to the controller communication processes, three other processes for concurrency control, directory management and record processing and, finally, a newly added disk I/O process to handle the disk operations.

B. THE SUMMARY OF THE WORK

The purpose of this thesis has been to examine the portability of MBDS. To this end we recommend a way to minimize the operating-system dependencies for a highly portable MBDS. In so doing, we have examined three areas in MBDS where portability is most likely to be compromised. These areas are: 1) the user interface, 2) the message-passing interface, and 3) the disk I/O interface.

First, with regard to the user interface, we have designed and developed the multiple record templates to allow the database to serve a variety of applications. For this development, it has been necessary to modify the internal template structure of the MBDS to support the new design. Extensive modifications have also been made to the user interface process in the controller. Included in the modifications are the complete restructuring of the database load modules, and the user modules for generating the template file, the descriptor file and the record files.

Second, we have downloaded the Ohio State version of the MBDS software that runs on the VAX and SUN configuration, to run on the eight ISI-workstations. In particular, we have examined the message-passing mechanisms from a high-level abstraction to the low-level support of the communication protocols. We have discussed the handling of the communications of two processes in the controller, of two processes in a backend, and of a process in the controller and a process in a backend. In addition, we have added broadcasting to allow the controller to send a message to all the backends and for a backend to send a message to all the other backends.

Finally, we have designed and developed the disk I/O process as a new process to the backend. We have considered three design alternatives and selected the backend process approach on the basis of its portability and its conformity with

our MBDS design. The disk I/O process is designed to provide efficient disk I/O operations. It includes such features as the common message interface. the real-time disk access, the asynchronous I/O and the effective disk access strategy.

C. PORTABILITY ISSUES AND SOLUTIONS

In chapter I. we have discussed the software portability issues and introduced the notion of the abstraction and isolation as a means of dealing with hardware and operating-system dependencies. For MBDS. we have identified two areas that are operating-system dependent. They are the message-passing interface and the disk-I/O interface. The message-passing interface as described in chapter III provides the required communications. It is operating-system dependent because the communication protocols. that support the message-passing interface, are operating-system dependent. The disk-I/O interface discussed in chapter IV provides the I/O operations. It is also operating-system dependent because the device driver that supports the disk I/O interface is a part of the operating system. To minimize the changes which we have to make in porting from one system to another, we provided each a high-level abstraction. For the message-passing interface. we have developed two routines:

```
send(message);
receive(message);
```

These routines, in turn, make appropriate system calls to deliver the message. For the disk-I/O interface, we have provided the do-disk-io routine with appropriate arguments to handle the disk-I/O operations. If the operating environment should change, we need only to rewrite these three routines.

For the design of future database systems, the real issue in portability is to be aware of hardware and operating-system dependencies, rather than to avoid them.

Once the portions of the database system that are hardware and operating-system dependent are identified. those portions should be made changeable. Portability of a database system can best be handled by making the system easy to change, rather than easy to port without any change. The former will result in a more efficient system. the latter will not.

APPENDIX A

THE USER INTERFACE

In this appendix we present a thorough walk through the user interface. It includes the database load operations as discussed in chapter II. and the execution of requests using existing files. The examples are also included for demonstrations of building requests and input files such as template, descriptor,and record files on-line.

```
How many backends are there? (1,2,...)>7


Do you want de-bugging messages printed? (y/n)>y
What operation would you like to perform?

      (g) - generate database
      (l) - load database
      (e) - execute test interface
      (x) - exit to operating system
      (z) - exit and Stop MDBS
l
What operation would you like to perform?

      (t) - load the template and descriptor files
      (r) - mass load a file of records
      (x) - exit, return to previous menu
t

ENTER NAME OF FILE CONTAINING TEMPLATE INFORMATION:

tt.f

ENTER NAME OF FILE CONTAINING DESCRIPTORS:

td.f
What operation would you like to perform?

      (t) - load the template and descriptor files
      (r) - mass load a file of records
      (x) - exit, return to previous menu
r
```

```
ENTER NAME OF FILE CONTAINING RECORDS TO BE LOADED:

tr20.f

TEST 001[INSERT(<TEMP,Part>,<PNO,P1>,<NAME,Idm>,<CITY,Mont>)]
TEST 002[INSERT(<TEMP,Part>,<PNO,P2>,<NAME,Xyz>,<CITY,Sali>)]
TEST 003[INSERT(<TEMP,Part>,<PNO,P3>,<NAME,Nut>,<CITY,Colu>)]
TEST 004[INSERT(<TEMP,Sups>,<SNO,S1>,<NAME,Nut>)]
TEST 005[INSERT(<TEMP,Sups>,<SNO,S2>,<NAME,Nut>)]
TEST 006[INSERT(<TEMP,Sups>,<SNO,S2>,<NAME,Nut>)]
TEST 007[INSERT(<TEMP,Sups>,<SNO,S1>,<NAME,Dec>)]
TEST 008[INSERT(<TEMP,Sups>,<SNO,S3>,<NAME,Nut>)]
TEST 009[INSERT(<TEMP,Sups>,<SNO,S3>,<NAME,Dec>)]
TEST 010[INSERT(<TEMP,Sups>,<SNO,S4>,<NAME,Nut>)]
TEST 011[INSERT(<TEMP,Sups>,<SNO,S4>,<NAME,Dec>)]
TEST 012[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,500>)]
TEST 013[INSERT(<TEMP,Ship>,<SNO,S2>,<PNO,P2>,<QTY,500>)]
TEST 014[INSERT(<TEMP,Ship>,<SNO,S3>,<PNO,P1>,<QTY,500>)]
TEST 015[INSERT(<TEMP,Ship>,<SNO,S4>,<PNO,P2>,<QTY,1000>)]
TEST 016[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,1000>)]
TEST 017[INSERT(<TEMP,Ship>,<SNO,S2>,<PNO,P2>,<QTY,1000>)]
TEST 018[INSERT(<TEMP,Ship>,<SNO,S3>,<PNO,P1>,<QTY,2000>)]
TEST 019[INSERT(<TEMP,Ship>,<SNO,S4>,<PNO,P2>,<QTY,2000>)]
TEST 020[INSERT(<TEMP,Ship>,<SNO,S1>,<PNO,P2>,<QTY,2000>)]


What operation would you like to perform?

      (t) - load the template and descriptor files
      (r) - mass load a file of records
      (x) - exit, return to previous menu
x
What operation would you like to perform?

      (g) - generate database
      (l) - load database
      (e) - execute test interface
      (x) - exit to operating system
      (z) - exit and Stop MDBS
e
Do you ALWAYS want to wait for responses? (y/n)
    > y
```

```
Enter the type of subsession you want
    (r) REDIRECT OUTPUT: select output for answers
    (d) NEW DATABASE: choose a new database
    (n) NEW LIST: create a new list of traffic units
    (m) MODIFY: modify an existing list of traffic units
    (s) SELECT; select traffic units from an existing list
            (or give new traffic units) for execution
    (o) OLD LIST: execute all the traffic units in an
            existing list
    (p) PERFORMANCE TESTING
    (x) EXIT: return to generate,load,execute, or exit menu

    SELECTION> s

    Enter the name for the traffic unit file
    It may be up to 13 characters long including the .ext.
    Filenames may include only one '#' character
    as the first character before the version number.

    File name> tRCreq.f
List of executable traffic units
 (0)[RETRIEVE(TEMP=Sups)(SNO,NAME)
        COMMON(SNO,SNO)
        RETRIEVE(TEMP=Ship)(PNO,SNO,QTY)]

 (1)[RETRIEVE(TEMP=Sups)(SNO,NAME)]

 (2)[RETRIEVE(TEMP=Ship)(PNO,SNO,QTY)]

 (3)[RETRIEVE(TEMP=Part)(PNO,NAME)
        COMMON(PNO,PNO)
        RETRIEVE(TEMP=Ship)(SNO,QTY)]

 (4)[RETRIEVE(TEMP=Part)(PNO,NAME)]

 (5)[RETRIEVE(TEMP=Ship)(SNO,QTY,PNO)
        COMMON(QTY,QTY)
        RETRIEVE(TEMP=Ship)(SNO,PNO)]

 (6)[RETRIEVE(TEMP=Ship)(SNO,QTY,PNO)]
```

```
/* This section shows how to invoke a predefined retrieve-common request */

    Select Options

    (d)  display the traffic units in the list
    (n)  enter a new traffic unit to be executed
    (num)    execute the traffic unit at [num]
    (x)  exit from this SELECT subsession

    Option> 0


TEST  023[RETRIEVE(TEMP=Sups)(SNO,NAME)
         COMMON(SNO.SNO)
         RETRIEVE(TEMP=Ship)(PNO,SNO,QTY)]


<COMMON.File> <SNO.S1> <NAME.Dec> <PNO.P2> <SNO,S1> <QTY.2000>
<COMMON.File> <SNO.S1> <NAME.Dec> <PNO,P2> <SNO,S1> <QTY,1000>
<COMMON,File> <SNO.S1> <NAME.Nut> <PNO,P2> <SNO,S1> <QTY,500>
<COMMON.File> <SNO.S1> <NAME.Nut> <PNO.P2> <SNO,S1> <QTY.2000>
<COMMON.File> <SNO.S1> <NAME.Dec> <PNO.P2> <SNO.S1> <QTY,500>
<COMMON.File> <SNO.S3> <NAME.Nut> <PNO,P1> <SNO.S3> <QTY,2000>
<COMMON.File> <SNO.S1> <NAME,Nut> <PNO.P2> <SNO.S1> <QTY,1000>
<COMMON.File> <SNO.S2> <NAME.Nut> <PNO.P2> <SNO,S2> <QTY,500>
<COMMON.File> <SNO.S3> <NAME.Nut> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON.File> <SNO.S3> <NAME.Dec> <PNO,P1> <SNO,S3> <QTY,2000>
<COMMON,File> <SNO.S2> <NAME.Nut> <PNO.P2> <SNO,S2> <QTY,1000>
<COMMON.File> <SNO.S2> <NAME.Nut> <PNO.P2> <SNO,S2> <QTY,500>
<COMMON.File> <SNO.S3> <NAME.Dec> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON,File> <SNO,S4> <NAME,Nut> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON.File> <SNO.S2> <NAME.Nut> <PNO,P2> <SNO,S2> <QTY,1000>
<COMMON.File> <SNO,S4> <NAME,Nut> <PNO,P2> <SNO,S4> <QTY,1000>
<COMMON.File> <SNO.S4> <NAME.Dec> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON.File> <SNO.S4> <NAME.Dec> <PNO.P2> <SNO,S4> <QTY,1000>
```

/* This section describes the method to build a retrieve-common request*/

Select Options

(d) display the traffic units in the list
(n) enter a new traffic unit to be executed
(num) execute the traffic unit at [num]
(x exit from this SELECT subsession

Option> n

Enter the character for the desired Traffic Unit type.

(r) Request
(t) Transaction (multiple requests)
(f) Finished entering traffic units.

Letter> r

Enter the character for the desired next step.

(i) INSERT
(r) RETRIEVE
(u) UPDATE
(d) DELETE
(c) RETRIEVE COMMON

LETTER> c

RETRIEVE COMMON Request
First enter the source retrieve request

RETRIEVE Request

Enter responses as you are prompted. You will be prompted first for the predicates of the query. then attributes for the target-list. next for an attribute for the optional BY clause and finally for a pointerfor the optional WITH clause.

When you have finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE> TEMP

Enter the character for the desired relational operator

(a) = EQUAL
(b) /= NOT EQUAL
(c) > GREATER THAN
(d) >= GREATER THAN or EQUAL
(e) < LESS THAN
(f) <= LESS THAN or EQUAL

Letter> a

Value> Sups

So far your conjunction is
(TEMP=Sups).
Do you wish to 'and' additional predicates to this conjunction? (y/n)
n
Do you wish to append more conjunctions to the query? (y/n)
n

Begin entering attributes for the Target-List. When you are through entering attributes respond to the ATTRIBUTE> prompt with <return>.
Do you wish to be prompted for aggregation?
n

ATTRIBUTE> SNO

ATTRIBUTE> NAME

ATTRIBUTE>

COMMON ATTRIBUTE 1> SNO

COMMON ATTRIBUTE 2> SNO

The request being built is:
[RETRIEVE(TEMP=Sups)(SNO,NAME)COMMON(SNO,SNO) 4

Enter the target retrieve

RETRIEVE Request

Enter responses as you are prompted. You will be prompted first for the predicates of the query, then attributes for the target-list. next for an attribute for the optional BY clause and finally for a pointerfor the optional WITH clause.

When you have finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE> TEMP
Enter the character for the desired relational operator

(a) = EQUAL
(b) /= NOT EQUAL
(c) > GREATER THAN
(d) >= GREATER THAN or EQUAL
(e) < LESS THAN
(f) <= LESS THAN or EQUAL

Letter> a

Value> Ship

So far your conjunction is
(TEMP=Ship).
Do you wish to 'and' additional predicates to this conjunction? (y/n)
n
Do you wish to append more conjunctions to the query? (y/n)
n

Begin entering attributes for the Target-List. When you are through entering attributes respond to the ATTRIBUTE> prompt with <return>.
Do you wish to be prompted for aggregation?
n

ATTRIBUTE> SNO

ATTRIBUTE> PNO

ATTRIBUTE> QTY

ATTRIBUTE>

```
The request being processed is:
[RETRIEVE(TEMP=Sups)(SNO.NAME)
   COMMON(SNO.SNO)
   RETRIEVE(TEMP=Ship)(SNO.PNO,QTY)]


<COMMON,File> <SNO,S1> <NAME,Dec> <PNO,P2> <SNO,S1> <QTY,2000>
<COMMON,File> <SNO,S1> <NAME,Dec> <PNO,P2> <SNO,S1> <QTY,1000>
<COMMON,File> <SNO,S1> <NAME,Nut> <PNO,P2> <SNO,S1> <QTY,500>
<COMMON,File> <SNO,S1> <NAME,Nut> <PNO,P2> <SNO,S1> <QTY,2000>
<COMMON,File> <SNO,S1> <NAME,Dec> <PNO,P2> <SNO,S1> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Nut> <PNO,P1> <SNO,S3> <QTY,2000>
<COMMON,File> <SNO,S1> <NAME,Nut> <PNO,P2> <SNO,S1> <QTY,1000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Nut> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Dec> <PNO,P1> <SNO,S3> <QTY,2000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,1000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,500>
<COMMON,File> <SNO,S3> <NAME,Dec> <PNO,P1> <SNO,S3> <QTY,500>
<COMMON,File> <SNO,S4> <NAME,Nut> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON,File> <SNO,S2> <NAME,Nut> <PNO,P2> <SNO,S2> <QTY,1000>
<COMMON,File> <SNO,S4> <NAME,Nut> <PNO,P2> <SNO,S4> <QTY,1000>
<COMMON,File> <SNO,S4> <NAME,Dec> <PNO,P2> <SNO,S4> <QTY,2000>
<COMMON,File> <SNO,S4> <NAME,Dec> <PNO,P2> <SNO,S4> <QTY,1000>

    Select Options

    (d)  display the traffic units in the list
    (n)  enter a new traffic unit to be executed
    (num)    execute the traffic unit at [num]
    (x)  exit from this SELECT subsession

    Option> x

Enter the type of subsession you want
    (r)  REDIRECT OUTPUT: select output for answers
    (d)  NEW DATABASE: choose a new database
    (n)  NEW LIST: create a new list of traffic units
    (m)  MODIFY: modify an existing list of traffic units
    (s)  SELECT: select traffic units from an existing list
              (or give new traffic units) for execution
    (o)  OLD LIST: execute all the traffic units in an
              existing list
    (p)  PERFORMANCE TESTING
    (x)  EXIT: return to generate,load.execute, or exit menu

    SELECTION> x
```

```
What operation would you like to perform?

      (g) - generate database
      (l) - load database
      (e) - execute test interface
      (x) - exit to operating system
      (z) - exit and Stop MDBS


g

Do you want de-bugging messages printed? (y/n)
y
What operation would you like to perform?

      (t) - generate record template
      (d) - generate descriptors
      (m) - generate/modify sets
      (r) - generate records
      (q) - quit, return to previous menu
            to load, execute or exit system
t

ENTER THE NAME OF THE FILE TO BE USED TO STORE
TEMPLATE INFORMATION:
wt.f

File 'wt.f' opened successfully

ENTER DATABASE ID:
TEST

ENTER THE NUMBER OF TEMPLATES FOR DATABASE TEST:
3

ENTER THE NUMBER OF ATTRIBUTES FOR TEMPLATE #1:
4

ENTER THE NAME OF TEMPLATE #1:
Part

ENTER ATTRIBUTE NAME #1 FOR TEMPLATE Part:
TEMP

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER ATTRIBUTE NAME #2 FOR TEMPLATE Part:
PNO

ENTER VALUE TYPE: (s=string, i=integer)
```

```
s

ENTER ATTRIBUTE NAME #3 FOR TEMPLATE Part:
NAME

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER ATTRIBUTE NAME #4 FOR TEMPLATE Part:
CITY

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER THE NUMBER OF ATTRIBUTES FOR TEMPLATE #2:
3

ENTER THE NAME OF TEMPLATE #2:
Sups

ENTER ATTRIBUTE NAME #1 FOR TEMPLATE Sups:
TEMP

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER ATTRIBUTE NAME #2 FOR TEMPLATE Sups:
SNO

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER ATTRIBUTE NAME #3 FOR TEMPLATE Sups:
NAME

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER THE NUMBER OF ATTRIBUTES FOR TEMPLATE #3:
4

ENTER THE NAME OF TEMPLATE #3:
Ship[

ENTER ATTRIBUTE NAME #1 FOR TEMPLATE Ship:
TEMpP

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER ATTRIBUTE NAME #2 FOR TEMPLATE Ship:
```

```
SNO

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER ATTRIBUTE NAME #3 FOR TEMPLATE Ship:
PNO

ENTER VALUE TYPE: (s=string, i=integer)
s

ENTER ATTRIBUTE NAME #4 FOR TEMPLATE Ship:
QTY

ENTER VALUE TYPE: (s=string, i=integer)
i
file 'wt.f' successfully closed
What operation would you like to perform?

      (t) - generate record template
      (d) - generate descriptors
      (m) - generate/modify sets
      (r) - generate records
      (q) - quit, return to previous menu
            to load, execute or exit system
d

ENTER THE NAME OF TEMPLATE FILE:
wt.f

file 'wt.f' opened successfully

ENTER THE NAME OF THE FILE TO BE USED FOR
STORING DESCRIPTORS:
wd.f

file 'wd.f' opened successfully

Do you want attribute 'TEMP' to be a directory attribute? (y/n)
y

ENTER THE DESCRIPTOR TYPE FOR TEMP:(A,B,C)
C

ENTER UPPER BOUND FOR EACH DESCRIPTOR IN TURN
-- ENTER '@' TO STOP

UPPER BOUND>
Sups

UPPER BOUND>
```

Ship

UPPER BOUND>
@

Do you want attribute 'PNO' to be a directory attribute? (y/n)
y

ENTER THE DESCRIPTOR TYPE FOR PNO:(A,B,C)
A

Use '!' to indicate that no lower bound exists ... Enter '@' to stop

ENTER LOWER BOUND FOR DESCRIPTOR:
!

ENTER UPPER BOUND FOR DESCRIPTOR: ... (lower bound = !)
P1

ENTER LOWER BOUND FOR DESCRIPTOR:
!

ENTER UPPER BOUND FOR DESCRIPTOR: ... (lower bound = !)
P2

ENTER LOWER BOUND FOR DESCRIPTOR:
@

Do you want attribute 'NAME' to be a directory attribute? (y/n)
y

ENTER THE DESCRIPTOR TYPE FOR NAME:(A,B,C)
A

Use '!' to indicate that no lower bound exists ... Enter '@' to stop

ENTER LOWER BOUND FOR DESCRIPTOR:
A

ENTER UPPER BOUND FOR DESCRIPTOR: ... (lower bound = A)
L

ENTER LOWER BOUND FOR DESCRIPTOR:
M

ENTER UPPER BOUND FOR DESCRIPTOR: ... (lower bound = M)
N

ENTER LOWER BOUND FOR DESCRIPTOR:

```
O

ENTER UPPER BOUND FOR DESCRIPTOR: ... (lower bound = O)
Z

ENTER LOWER BOUND FOR DESCRIPTOR:
@

Do you want attribute 'CITY' to be a directory attribute? (y/n)
n

Do you want attribute 'SNO' to be a directory attribute? (y/n)
n

Do you want attribute 'QTY' to be a directory attribute? (y/n)
y

ENTER THE DESCRIPTOR TYPE FOR QTY:(A,B,C)
A


Use '!' to indicate that no lower bound exists ... Enter '@' to stop

ENTER LOWER BOUND FOR DESCRIPTOR:
1

ENTER UPPER BOUND FOR DESCRIPTOR: ... (lower bound = 1)
1000

ENTER LOWER BOUND FOR DESCRIPTOR:
1000

ENTER UPPER BOUND FOR DESCRIPTOR: ... (lower bound = 1000)
9000

ENTER LOWER BOUND FOR DESCRIPTOR:
@

file 'wt.f' successfully closed
file 'wd.f' successfully closed
What operation would you like to perform?

      (t) - generate record template
      (d) - generate descriptors
      (m) - generate/modify sets
      (r) - generate records
      (q) - quit, return to previous menu
            to load, execute or exit system
r

ENTER THE NAME OF TEMPLATE FILE:
```

```
wr.f

**ERROR** cannot open template file 'wr.f'

RE-ENTER TEMPLATE FILE NAME:
wt.f

file 'wt.f' opened successfully

ENTER THE NAME OF THE FILE TO BE USED FOR
STORING RECORDS:
wr.f

file 'wr.f' opened successfully

ENTER THE NAME OF THE FILE CONTAINING THE VALUES
FOR ATTRIBUTE
PNO.s

file 'PNO.s' opened successfully
file 'PNO.s' successfully closed

ENTER THE NAME OF THE FILE CONTAINING THE VALUES
FOR ATTRIBUTE
NAME.s

file 'NAME.s' opened successfully
file 'NAME.s' successfully closed

ENTER THE NAME OF THE FILE CONTAINING THE VALUES
FOR ATTRIBUTE
CITY.s

file 'CITY.s' opened successfully
file 'CITY.s' successfully closed

48 records can be generated for template 'Part'...

How many records do you want generated?
6

ENTER THE NAME OF THE FILE CONTAINING THE VALUES
FOR ATTRIBUTE
SNO.s

file 'SNO.s' opened successfully
file 'SNO.s' successfully closed

ENTER THE NAME OF THE FILE CONTAINING THE VALUES
FOR ATTRIBUTE
NAME.s
```

file 'NAME.s' opened successfully
file 'NAME.s' successfully closed

28 records can be generated for template 'Sups'...

How many records do you want generated?
4

ENTER THE NAME OF THE FILE CONTAINING THE VALUES FOR ATTRIBUTE
SNO.ds

file 'SNO.s' opened successfully
file 'SNO.s' successfully closed

ENTER THE NAME OF THE FILE CONTAINING THE VALUES FOR ATTRIBUTE
PNO.s

file 'PNO.s' opened successfully
file 'PNO.s' successfully closed

ENTER THE NAME OF THE FILE CONTAINING THE VALUES FOR ATTRIBUTE
QTY.s

file 'QTY.s' opened successfully
file 'QTY.s' successfully closed

28 records can be generated for template 'Ship'...

How many records do you want generated?
2

****

ALL RECORDS GENERATED

****

file 'wt.f' successfully closed
file 'wr.f' successfully closed
What operation would you like to perform?

(t) - generate record template
(d) - generate descriptors
(m) - generate/modify sets
(r) - generate records
(q) - quit, return to previous menu
to load, execute or exit system

```
m

ENTER THE NAME OF TEMPLATE FILE:
wt.f

file 'wt.f' opened successfully

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'TEMP' ON TEMPLATE 'Part':
      (n) - generate a new set for it
      (m) - modify an existing set for it
      (s) - do nothing with it
n

ENTER THE NAME OF THE FILE TO BE USED TO STORE THE SET:
w.s

file 'w.s' opened successfully

ENTER SET VALUE:
J1

ENTER SET VALUE:
J2

ENTER SET VALUE:
@

file 'w.s' successfully closed
Set generation completed ... do you want to modify it ?
y

ENTER THE FILE NAME OF THE SET TO BE MODIFIED
w.s

file 'w.s' opened successfully

What function do you want to perform next?
      (p) - print the set elements and their indices
      (a) - add some elements to the set
      (r) - remove some elements from the set
      (n) - nothing: done
p
  INDEX    ELEMENT
   0        J1
   1        J2

What function do you want to perform next?
      (p) - print the set elements and their indices
      (a) - add some elements to the set
      (r) - remove some elements from the set
```

```
     (n) - nothing: done
n

Do you want to store the modified set back into the original file ? (y/n)
n

ENTER THE NAME OF THE FILE TO BE USED TO STORE THE SET:
@

file '@' successfully closed

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'PNO' ON TEMPLATE 'Part':
     (n) - generate a new set for it
     (m) - modify an existing set for it
     (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'NAME' ON TEMPLATE 'Part':
     (n) - generate a new set for it
     (m) - modify an existing set for it
     (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'CITY' ON TEMPLATE 'Part':
     (n) - generate a new set for it
     (m) - modify an existing set for it
     (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'TEMP' ON TEMPLATE 'Sups':
     (n) - generate a new set for it
     (m) - modify an existing set for it
     (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'SNO' ON TEMPLATE 'Sups':
     (n) - generate a new set for it
     (m) - modify an existing set for it
     (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'NAME' ON TEMPLATE 'Sups':
     (n) - generate a new set for it
     (m) - modify an existing set for it
     (s) - do nothing with it
```

```
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'TEMP' ON TEMPLATE 'Ship':
      (n) - generate a new set for it
      (m) - modify an existing set for it
      (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'SNO' ON TEMPLATE 'Ship':
      (n) - generate a new set for it
      (m) - modify an existing set for it
      (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'PNO' ON TEMPLATE 'Ship':
      (n) - generate a new set for it
      (m) - modify an existing set for it
      (s) - do nothing with it
s

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'QTY' ON TEMPLATE 'Ship':
      (n) - generate a new set for it
      (m) - modify an existing set for it
      (s) - do nothing with it
s
file 'wt.f' successfully closed
What operation would you like to perform?

      (t) - generate record template
      (d) - generate descriptors
      (m) - generate/modify sets
      (r) - generate records
      (q) - quit, return to previous menu
            to load, execute or exit system
q
What operation would you like to perform?

      (g) - generate database
      (l) - load database
      (e) - execute test interface
      (x) - exit to operating system
      (z) - exit and Stop MDBS


z
```

APPENDIX B

The Program Specifications of the Disk I/O Process

This appendix contains a detail specification for the design of the disk I/O process. The specification is written in a high-level system specification language which should be readable. The language has typical constructs:

1) perform procedure (); /* a procedure call */
2) if expression then statements else statement endif
3) while condition do statements endwhile
4) the condition do statements endif
5) case type 'constant' statements endcase

The language uses a dot notion to number each statement. The number between the dots, for example, 10.1 indicating statement 10 in the main program, is a performance or procedure call. The procedure is defined beginning with number 10.1. There can be many levels or dots.

Data Structures:

```
Structure gue-infor {
        sender: integer          /* name of sender */
        request-id: string       /* request code */
        address: array           /* disk, cylinder and track number */
        op code: integer         /* read or write operation */
        link: pointer            /* link to next que-info */
        n block: integer         /* number of block in a single request*/
        chain: pointer           /* que-info chain for multiple
                                    block request */
        chain-head: pointer      /* head of que-info chain */
        block-ptr:               /* pointer to the data block */
}
que-info head: pointer           /* head of link list */
process-que-head: array          /* array of pointer for processing
                                    que, one for each disk */
io-entry: array                  /* selected entries one from each
                                    processing queue */
drive-head: array                /* position of disk arm, one for each
                                    disk */
```

Program Specifications:

```
1         task disk i/o
2         perform DIO-init();
3         Stop := False;
4         time := o;
5         while Stop is not time do
6             perform get-message ();
7             if message is received
8                 then
9                     if message is not a stop message
10                        then perform que-message ();
11                            if message buffer is full
12                                then
13                                    perform process();
14                                    perform get-entry();
15                                    perform do-io();
16                                else
17                                    time := time +1;
18                            endif
19                    else
20                        stop : TRUE;
21                endif
22            else
23                if timeout
24                    then
25                        perform process();
26                        perform get-entry();
27                        perform do-io ();
28                    else
29                        time := time+1
30                endif
31            endif
32        endwhile
33        endtask
```

```
6.1     proc get-message()
6.2        receive message
6.3        if message is received
6.4           then
6.5              return (TRUE);
6.6           else
6.7              return (FALSE);
6.8        endif
6.9     endproc
```

```
10.1    proc que-message
10.2       allocate storage for que-info
10.3       parse message into que-info
10.4       if message is a write request
10.5          then
10.6             allocate storage for data block
10.7             copy block point in que-info
10.8             push que-info pointer in stack
10.9             msg := msg+1;
10.10            return
10.11         else
10.12            get number of address
10.13            for each address do
10.14               allocate storage for data block
10.15               copy que-info from previous que info pointer
10.16               copy next address from message buffer
10.17               update previous pointer
10.18               push current info pointer in stack
10.19               msg := msg+1;
10.20               update previous que-info pointer
10.21            end for
10.22       endif
10.23    endproc
```

```
13.1    proc process ()
13.2       message := que-info-head:
13.3       if message is NULL
13.4          then
13.5             return ();
13.6       endif
13.7       while message pointer is not NULL do
13.8          get process-head for the disk in message
13.9          perform sortmerge (message, process-head);
14.0          update message to next link
14.1       end while
14.2       que-info head = NULL
14.3    end proc
```

```
13.9.1      proc sortmerge (message, process
13.9.2        if no element in process-que head
13.9.3          then
13.9.4            replace process-head with message pointer
13.9.5            return ();
13.9.6        endif
13.9.7        computer track number from address element for message
13.9.8        while process-que is not empty do
13.9.9          computer track number for process element
13.9.10         if process track number is greater than that of
                message
13.9.11           then
13.9.12             if process pointer is the head
13.9.13               then
13.9.14                     insert message pointer as process-head
13.9.15               else
13.9.16                 insert message pointer
13.9.17             endif
13.9.18           else
13.9.19             get next pointer in process list
13.9.20         endif
13.9.21       endwhile
13.9.22       attach message pointer to end of process list
13.9.23     endproc
```

```
            /* This routine select a process element closest to the
               current head position in the forward direction */
14.1        proc get-entry ()
14.2           for each disk do
14.3              get process-head queue for that disk
14.4              if process-head queue has no element
14.5                 then
14.6                       io-entry of that disk := NULL;
14.7                       continue to next disk
14.8              endif
14.9              while process list is not empty do
14.10                computer track number
14.11                if track is less the current head position
14.12                   then
14.13                      get next element in list
14.14                   else
14.15                      io-entry of that disk = process pointer
14.16                      drive-position = track number
14.17                endif
14.18          endfor
14.19       endproc
```

```
15.1        proc do-io
15.2           for each disk do
15.3              if io-entry of that disk is NULL
15.4                 then
15.5                    skip to the next disk
15.6              endif
15.7              get value of drive-position for that disk
15.8              position drive
15.9              case opcode
15.10                'READ'
15.11                   get block-pointer from que-info
15.12                   read block data
15.13                   get chain-head from que-info
15.14                   while chain list is empty do
15.15                      if block is read
15.16                         then
15.17                            get next pointer in chain
15.18                         else
15.19                            return ();
15.20                      endif
15.21                   endwhile
15.22                   perform io-send(io-entry):
15.23                   free all structures and blocks
15.24                   break:
15.25                'WRITE'
15.26                   get block-pointer
15.27                   write data block
15.28                   io-send(io-entry);
15.29                   free structure and block
15.30                   break;
15.31             endcase
15.32          endfor
15.33       endproc
```

```
15.22.1     io-send (io-entry)
15.22.2        get opcode from io-entry pointer to que-info structure
15.22.3        case opcode
15.22.4           'READ'
15.22.5              fill message buffer with request-id
15.22.6              fill message buffer with addresses
15.22.7              fill message buffer with data blocks
15.22.8              set message type to PIO-READ
15.22.9              return message to sender
15.22.10             break:
15.22.11          'WRITE'
15.22.12             fill message buffer with request-id
15.22.13             fill message buffer with address
15.22.14             set message type to PIO-WRITE
15.22.15             return message to sender
15.22.16             break;
15.22.17       endcase
15.22.18    endproc
```

LIST OF REFERENCES

1. Hsiao, D. K., and Menon, M. J., Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (part I). *Naval Postgraduate School Technical* Report *nps52-83-006*, 1983.

2. Hsiao, D. K., and Menon, M. J., Design and Analysis of a Multi-Backend Database System for Performance Improvement. Functionality Expansion and Capacity Growth (part II), *Naval Postgraduate School Technical* Report *nps52-83-007*, 1983.

3. Kerr, D.S., Orooji, A., Shi, Z. and Strawser, P. R., The implementation of a multi-backend Database System (MBDS): part I - Software Engineering Strategies and Efforts Towards a Prototype MBDS, *Naval Postgraduate School Technical Report nps52-83-008*. 1983.

4. He, X., Higashida, M., Hsiao, D. K., Kerr, D. S., Orooji, A., Shi, Z. and Strawser, P. R., The Implementation of a Multi-Backend Database System (MBDS): part II - The First Prototype MBDS and the Software Engineering Experience. *Naval Postgraduate School Technical Report nps52-82-008*, 1982.

5. Boyne, R. D.,Demurjian, S. A.,Hdsiao, D. K., Kerr, D. S. and Orooj, A., The Implementation of a Multi-Backend Database System (MBDS): part III - The message-Oriented Version with Concurrency Control and Secondary-Memory-Based Directory Management, *Naval Postgraduate School Technical* Report *nps52-83-003*, 1983.

6. Demurjian, S. A.,Hdsiao, D. K., Kerr, D. S. and Orooj, A., The Implementation of a Multi-Backend Database System (MBDS): part IV - The Revised Concurrency Control and Directory Management Processes and The Reviced Definitions of Inter-computer Message. *Naval Postgraduate School Technical Report nps52-84-005*, 1984.

7. Hsiao, D. K., A Generalized Record Organization, *IEEE Transactions On Computer C-20 Number 12*, 1971. sp 1

8. Hunt, A. L., The Implementation of The Primary Operation, Retrieve-Common of the Multi-Backend Database System (MBDS), *Master Thesis, Naval Postgraduate School, Monterey, CA*, 1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
2.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
3.	Curriculum Officer, Code 37 Computer Technology Programs Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
5.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
6.	Ablert Wong, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	6
7.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2

46
18070 2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5101

219664

Thesis
W755
c.1

Wong
Toward highly portable database systems: issues and solutions

11 JUN 93 38360

219664

Thesis
W755
c.1

Wong
Toward highly portable database systems: issues and solutions

thesW755
Toward highly portable database system:
3 2768 000 67968 2
DUDLEY KNOX LIBRARY